Table of Contents

Introduction

Model-View-Controller (MVC) is probably the most used architectural solution for User Interface design and web programming; Introduced first in the 70s, MVC has been progressively adapted and morphed into a wide range of subtypes and variations, so much that the plain term "MVC" without additional qualifications has lost specificity. As a general interpretation, it is a rather loose guideline for organizing your code when the data and visualization parts of your application need to interact while staying as loosely coupled as possible. How this is accomplished in practice depends on the particular MVC incarnation.

MVC can be seen as an aggregation of more fundamental design patterns such as Composite, Mediator and Observer/Notifier. The complexity and variation in style of MVC arises from all the possible uses and variations of these independent patterns to satisfy the potentially elaborate requirements of a GUI.

The objective of this book is to explore variations and nuances of MVC, comparing and analyzing them. The differentiating characteristic among them is to assign responsibilities to protagonists, specifically “who is responsible for what” and “who knows about whom” in the interaction between the User and the application state. MVC variations assign new and old responsibilities in different ways, connect or organize protagonists, or add intermediate objects to gain more flexibility and satisfy peculiar use cases.

This book is structured as follows:

- The first chapter will introduce a simple ground-up MVC application through code, with the objective of deploying a common vocabulary. The chapter will define components, roles, and communication patterns, and close with a remark on how the resulting formulation is outdated and too simplistic in modern software development.
- Once equipped with nomenclature, the second chapter will introduce MVC variations to address specific UI constraints and practical needs, or to improve development efficiency.
- The third chapter will expand the concept of MVC to hierarchical MVC schemes.
- The fourth chapter will focus on special techniques that emerge from a complex modern GUI.
- In the fifth and final chapter, we will specifically focus on Web MVC and its implementations.

Throughout the book, example code or actual implementations will be presented to clarify design ideas. GUI rendering will make use of the excellent Qt toolkit. Qt provides pre-made mechanisms to address some MVC needs, but in the upcoming code these mechanisms will be skipped on purpose to demonstrate the presented concepts.

Acknowledgements and motivations

I started writing this book as an accident. Initially, I wanted to write a series of blog posts to describe Model View Controller and a few related patterns. As I gathered more and more information from the net and my personal experience, I suddenly found out that the amount and structure of what I wrote was beyond the scope of a blog, hence the decision to re-label it as a book. I am happy with the decision, because it gave me freedom to add material I would not have added otherwise.

This works presents and enriches design solutions, best practices, and experiments made available by countless blog posts and comments. To these authors goes my acknowledgement and gratitude. Being a work in progress, there's still a lot to be done. Please be patient, but feel free to send me feedback, pull requests, and take advantage of the material already present.

This book is released under GFDL license, and free (gratis), mainly for three reasons

- As stated, most of the material here presented was gathered from the net. It was my personal effort to organize this knowledge, but I had a lower startup barrier.
- I already published a book with a commercial publisher, and from my experience and math, I think that if I put a book on the web and accept donations I would probably get more money and feedback than going through a publisher.
- This book is part of my portfolio as a professional in software development and design, and I am proud to focus on a personal project to increase my competences.

That said, I gladly accept donations:

- on GratiPay
- on bitcoin
- on PayPal (using my email address stefano.borini at ferrara dot linux dot it)

The sources of this book are available as a github repository at the following URL:

https://github.com/stefanoborini/modelviewcontroller-src

I also have a personal website at http://forthescience.org where you can find more information about me, my curriculum and activities.

Basics of MVC: From Smart-UI to Traditional MVC

The aim of GUI programming is to provide an interactive and updated visual representation of the current state of the application. A typical set of requirements is the need to simultaneously represent this state graphically in various forms (i.e. a table of numbers and an XY plot), modify this state through an unpredictable sequence of mouse/keyboard events, and keep the state and its visual representation always synchronized and up-to-date. At the same time, the application state must respect business logic constraints and be kept within these constraints. At the code level, these requirements are generally translated into objects interacting through a communication network of various degrees of complexity. How can we satisfy these requirements, while at the same time providing a flexible design that keeps the object communication simple, understandable and organized as much as possible?

MVC addresses the above needs. It does so by clever subdivision of competences and roles in the code, while introducing constraints that keep a potentially chaotic communication well organized and streamlined. MVC is incredibly flexible and adaptable, as we will see, and using one of the many styles will be a matter of preference or constraints/best practices of the development framework of choice.

GUI Programming is a complex task. Many different levels of understanding and handling are needed: UI design and usability consideration, multithreading and multiprocessing for asynchronous evaluation, event notification coherence and balancing, adaptability of the GUI to unexpected requests and changes of style. There are plenty of dialogs, buttons, lists, all with different performance, presentation and visibility needs. In a sense, a GUI application develops emerging properties characteristic of a complex system where multiple entities interact. Keeping this system under strict control is the only way to maintain chaos at bay.

In this chapter, we will start from the most trivial implementation of a GUI application with both visual and non-visual logic: a single class responsible for everything. This approach, known as Smart UI, will be our foundation for a progressive refactoring into the three basic components of MVC: Model, View and Controller. We will call the result a “traditional MVC” design.

Smart-UI: A single class with many responsibilities

We start this exploration toward MVC with the most trivial and simplistic design: **Smart UI**, also known as **Autonomous View**.

Important

A confusing characteristic of MVC literature is that different names are used to express the same concepts. Vice-versa, it is also common that the same name is used to express totally different concepts. We accept this by proposing the most common names, reporting "also known as" names, and stressing differences when appropriate.

The Smart UI approach uses a single class to handle all responsibilities we expect from a GUI program:

- Receives user driven events, such as mouse clicks and keyboard input
- Holds application logic to convert user driven events into changes of application state
- Holds the relevant application state
- Performs visual rendering of its state

As an example implementation of a Smart UI, consider a click counter application, which shows a button with a number. The number is increased every time the button is clicked.



The code is as follows:

```
import sys
from PyQt4 import QtCore, QtGui

class Counter(QtGui.QPushButton):
    def __init__(self, *args, **kwargs):
        super(Counter, self).__init__(*args, **kwargs)
        self._value = 0
        self._update()

    def mouseReleaseEvent(self, event):
        super(Counter, self).mouseReleaseEvent(event)
        self._value += 1
        self._update()

    def _update(self):
        self.setText(unicode(self._value))

app = QtGui.QApplication(sys.argv)
counter = Counter()
counter.show()
app.exec_()
```

The application's main and only visual component, `Counter`, is derived from a single GUI class, a Qt `QPushButton`. Observe in particular how `Counter` is

1. Storing the current click count value in the member variable `self._value`.
2. Handling the logic that modifies `self._value`. Every time the button is clicked, Qt invokes `mouseReleaseEvent` automatically. In this method the click counter is incremented.
3. Synchronizes the aspect of the button with the current `self._value`, by invoking `setText`.

This minimalist design seems appealing for its simplicity and compactness. It is a good starting point for trivial applications, and the one most likely to be implemented by novices in GUI programming, but it does not scale well for larger applications, where state, user events and graphic layout are more complex and intertwined and need to change often under development pressure. Specifically, observe the following issues:

- Access and modification of the current state from outside is cumbersome, being contained into the all-encompassing visual object: external objects that want to modify the current counter need to make sure that the represented value is synchronized, for example, forcing a call to `_update()`, or having the `Counter` object provide a `setValue()` method.

- It is difficult for other visual objects to report the same information, maybe with two different visual aspects (*e.g.* both as a counter and as a progress bar)

- The resulting class is difficult to test. The only way to stress it through its public interface and functionality is to actually probe it with GUI events, which is impractical for reasons we will examine later.
- The logic dealing with visual aspect (i.e. handling and layouting widgets, updating the label on the button), interaction aspect (handling the user initiated mouse click to perform the increment operation) and business aspect (incrementing the counter) are of different nature, and would be better kept separated. This would ease testability, simplify code understanding and interaction.

Document-View: dividing state from GUI

To solve the shortcomings of Smart UI, we take advantage of the intrinsic division into visual rendering, interaction and business logic expressed by a GUI application. In Smart UI, these three roles happen to be assigned to the same class, but we can reorganize our code so that the business logic part is kept separated. The resulting design is a two-class system known in literature as **Document View** or **Model Delegate**.

The **Document** class is responsible for handling the business logic. It has no part in dealing with graphical rendering, nor with GUI events. It simply stores application relevant state, and provides an interface to obtain this state or change it according to the rules of the application. Additionally, it provides a mechanism to inform interested objects of changes.

The **View** class instead handles user events, renders itself visually, performs operations on the Document and keeps its visual aspect synchronized against the Document's state when it changes.

The Document-View design achieves separation of the state from its graphical representation, allowing them to change independently. The Document has become a fully non-GUI entity that can act and be tested independently. Any registered View always keeps itself up-to-date against the Document contents through the notification system, and carry full responsibility for graphical rendering of the Document information and the handling of user interaction.

This design removes some of the concerns expressed for Smart UI. Testing of the state and business logic becomes easier: the Document object can be modified or accessed programmatically by issuing calls to its methods. This object is now independent and can work and manipulated with different Views, if desired. An additional price in complexity is introduced in having to keep the View (or Views) notified of changes to the Document.

Implementation example

We can implement this design to our Click counter application through progressive refactorings. The first step is to partition out the data, represented by the `self._value` variable, into the Document class. For our system to continue to work, the visual part View must now be informed of changes to this data. The Document will therefore not only hold `self._value`, but also provide an interface to query and modify this data and a strategy to notify other objects when changes occur. This is expressed in the following implementation code

```
class CounterDocument(object):
   def __init__(self):
       self._value = 0
       self._listeners = set()
```

In addition to the value, the `self._listeners` member variable holds references to Views interested in being notified about changes. We use a python set instead of a list to prevent accidental registration of the same object twice. Interested objects can register and unregister through the following methods

```
class CounterDocument(object):
  # ...
  def register(self, listener):
      self._listeners.add(listener)
      listener.notify()

  def unregister(self, listener):
      self._listeners.remove(listener)
```

We then provide a getter method [1] for `self._value` :

```
class CounterDocument(object):
   # ...
   def value(self):
       return self._value
```

We also provide a setter to directly set a specific value. Note in particular how the method notifies the registered listeners when the value changes. This is done by calling the listeners' `notify` method, as you can see in `self._notifyListeners`

```
class CounterDocument(object):
   # ...
   def setValue(self, value):
       if value != self._value:
           self._value = value
           self._notifyListeners()

   def _notifyListeners(self):
       for l in self._listeners:
           l.notify()
```

The method `notify` is therefore the interface that a registered listener must provide in order to receive notifications about the mutated state of the Document object. Our View need to implement this method.

Finally, we also provide a method that increments the value according to the expected logic

```
class CounterDocument(object):
   # ...
   def incrementValue(self):
       self._value += 1
       self._notifyListeners()
```

The View class will be responsible for rendering the information contained in an instance of `CounterDocument`. This instance is passed at initialization, and after a few formalities, the View register itself for notifications

```
class CounterView(QtGui.QPushButton):
   def __init__(self, document):
       super(CounterView, self).__init__()
       self._document = document
       self._document.register(self)
```

When this happens, the Document adds the View as a listener. A notification is immediately delivered to the newly added listener so that it can update itself. The `notify` method on the View is then called, which will query the current value from the Document, and update the text on the button

```
class CounterView(QtGui.QPushButton):
   # ...
   def notify(self):
       self.setText(unicode(self._document.value()))
```

Note how this method inquires the Document through its interface (calling `CounterDocument.value`). The View must therefore have detailed knowledge of its associated Model's interface and must deal with the semantic level it presents. Through this knowledge, the View extracts data from the Model, and converts “Model language” into “View language” to present the data into the visual widgets it is composed of.

Handling of the click event from the User is performed in `mouseReleaseEvent`, as in Smart-UI. This time however, the action will involve the Document, again through its interface

```
    class CounterView(QtGui.QPushButton):
        # ...
        def mouseReleaseEvent(self, event):
            super(CounterView, self).mouseReleaseEvent(event)
            self._document.incrementValue()
```

the `setValue` call will then issue a change notification that will update the button text via `notify`.

We can now provide multiple Views with different representation modes for the same information, or modify it through different sources, either visual or non-visual. We can for example add a Progress Bar

```
class ProgressBarView(QtGui.QProgressBar):
    def __init__(self, document):
        super(ProgressBarView, self).__init__()
        self._document = document
        self._document.register(self)
        self.setRange(0,100)

    def notify(self):
        self.setValue(self._document.value())
```

and register it on the same Document instance at initialization

```
app = QtGui.QApplication(sys.argv)

document = CounterDocument()
counter = CounterView(document)
progress = ProgressBarView(document)

counter.show()
progress.show()

app.exec_()
```

When the button is clicked, both its label and the progress bar are kept updated with the current value in the Document.

[1] Python properties can be used for the same goal. However, python properties are harder to connect to the signal/slots mechanism in PyQt.

[2] When registration of the View on the Document is done in the View's initializer, as we are doing here, it should be done only when the initialization is completed, so that notify can be called on a fully initialized object. An alternative strategy is to delay this setup and perform it through a `View.setDocument` method.

Notification system in strongly typed languages

A possible implementation of the notification system in strongly typed languages uses an interface class `ListenerInterface` with one abstract method `notify()`. For example, in C++ we could write the following code

```
class ListenerIface
{
public:
    virtual void notify() = 0;
};
```

Concrete listeners will implement this interface

```
class View : public ListenerIface
{
public:
    void notify();
};
```

The Model will accept and handle pointers to the Listener interface, thus not requiring a dependency toward specific Views or Controllers

```
class Model
{
public:
    void register(ListenerIface *listener)
    {
        listeners.push_back(listener);
    }

private:
    void notifyListeners()
    {
        std::vector<ListenerIface *>::iterator it;
        for (it = listeners.begin(); it != listeners.end(); ++it) {
                (*it)->notify();
    }

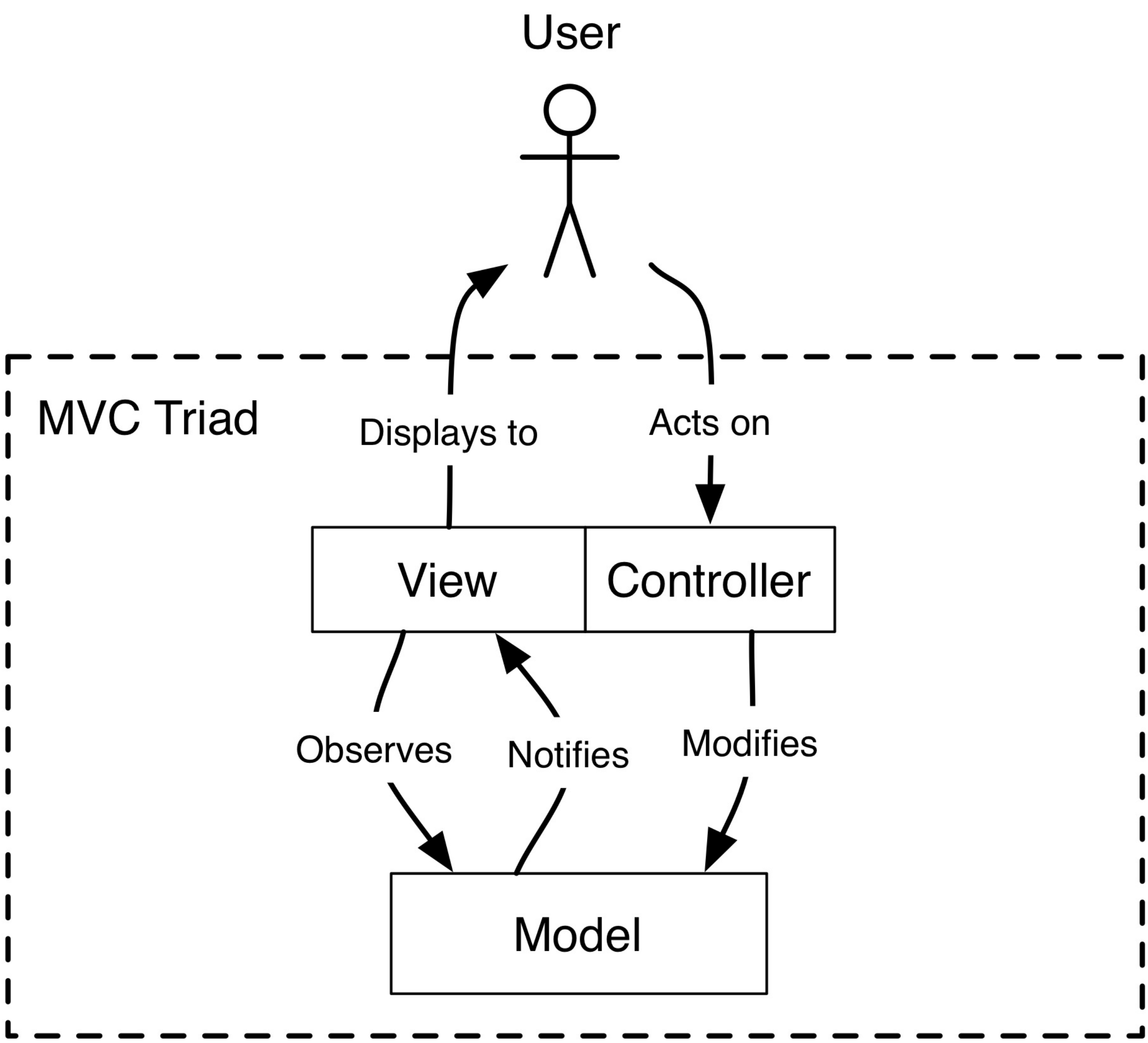
    std::vector<ListenerIface *> listeners;
};
```

A similar approach can be used in Java.

Traditional MVC

With the Document-View design we successfully extracted state from an initial Smart-UI design. The next objective is to extract the code that converts the primary event (in this case, a mouse click on the button) into the execution of the logic that modifies the state (addition of one to the value). The final result of this refactoring will be a **Traditional MVC** design[1].

In Traditional MVC, the Document is called **Model**, and its role and structure is unchanged: it stores state and delivers change notifications. The View part is divided into two classes, the **Controller** and the **View**. Once instantiated and connected, Model, View, and Controller form a so-called **MVC triad**.



The Controller's role is to transform primary events delivered by the View into operations on the Model. Depending on the specifics of the application, a Controller may or may not need a reference to the View, but it certainly needs the Model to apply changes on

```
class Controller(object):
   def __init__(self, model, view):
       self._model = model
       self._view = view
```

The method `addOne` performs the specific task of transforming a primary event into a Model operation, adding one to the current value. Obviously, the Controller does so through the Model interface. This operation will trigger a Model notification to its listeners

```
class Controller(object):
    # ...
    def addOne(self):
        self._model.setValue(self._model.value()+1)
```

At initialization, the View instantiates its associated Controller, passing itself and the Model as parameters. As before, the View registers itself on the Model via the `register` method

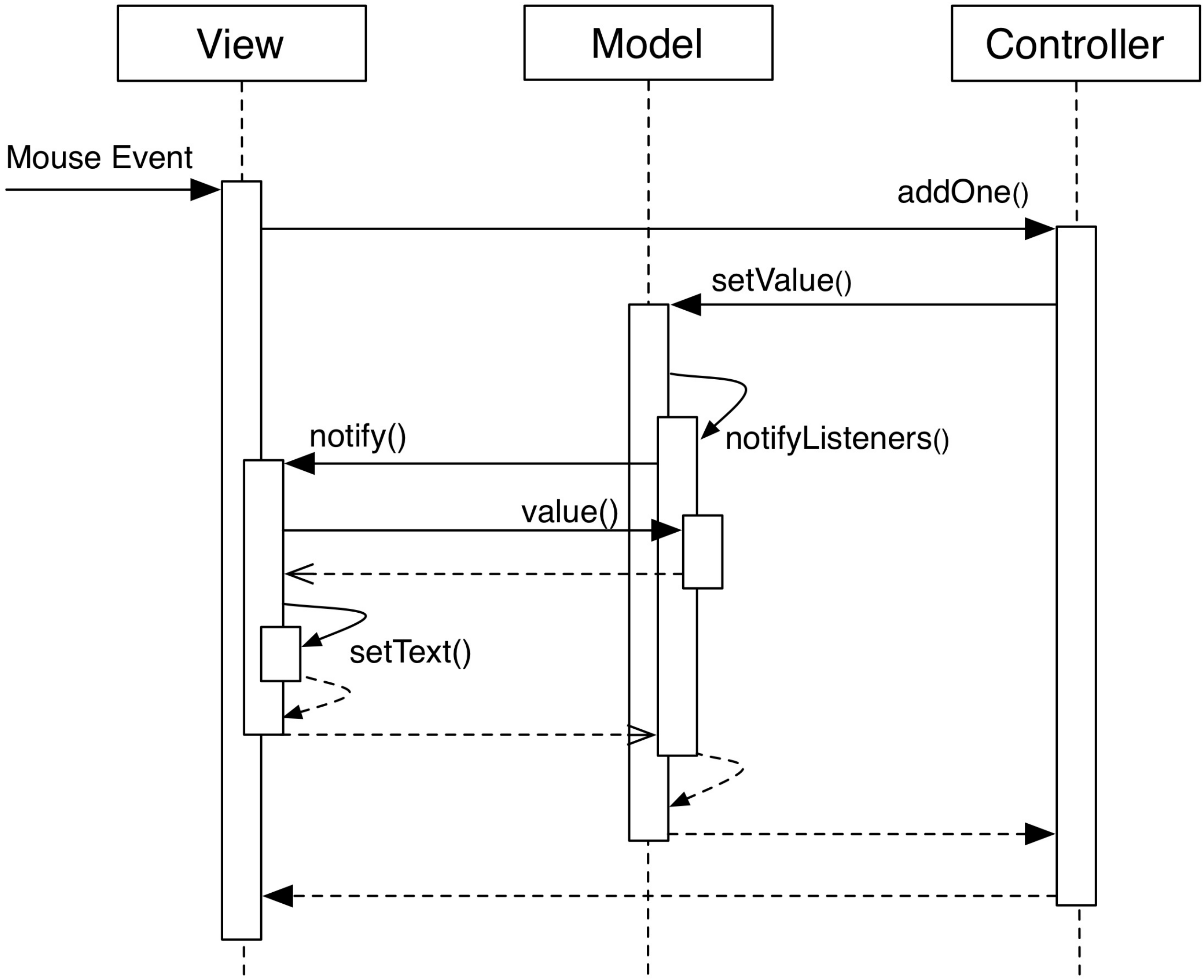
```
class View(QtGui.QPushButton):
    def __init__(self, model):
        super(View, self).__init__()
        self._model = model
        self._controller = Controller(self._model, self)
        self._model.register(self)
```

The View now depends on the Controller to modify the Model: only strictly GUI-related handling is done by the View. Conversion from GUI events to application business logic is delegated to the Controller in `mouseReleaseEvent`

```
class View(QtGui.QPushButton):
    # ...
    def mouseReleaseEvent(self, event):
        super(View, self).mouseReleaseEvent(event)
        self._controller.addOne()

    def notify(self):
        self.setText(unicode(self._model.value()))
```

Clicking on the View button will result in a call to `Controller.addOne`, in turn triggering a call to `notify` that updates the text label. The activity diagram shows the dance of calls presented above. Note how the Model-View synchronization does not involve the Controller



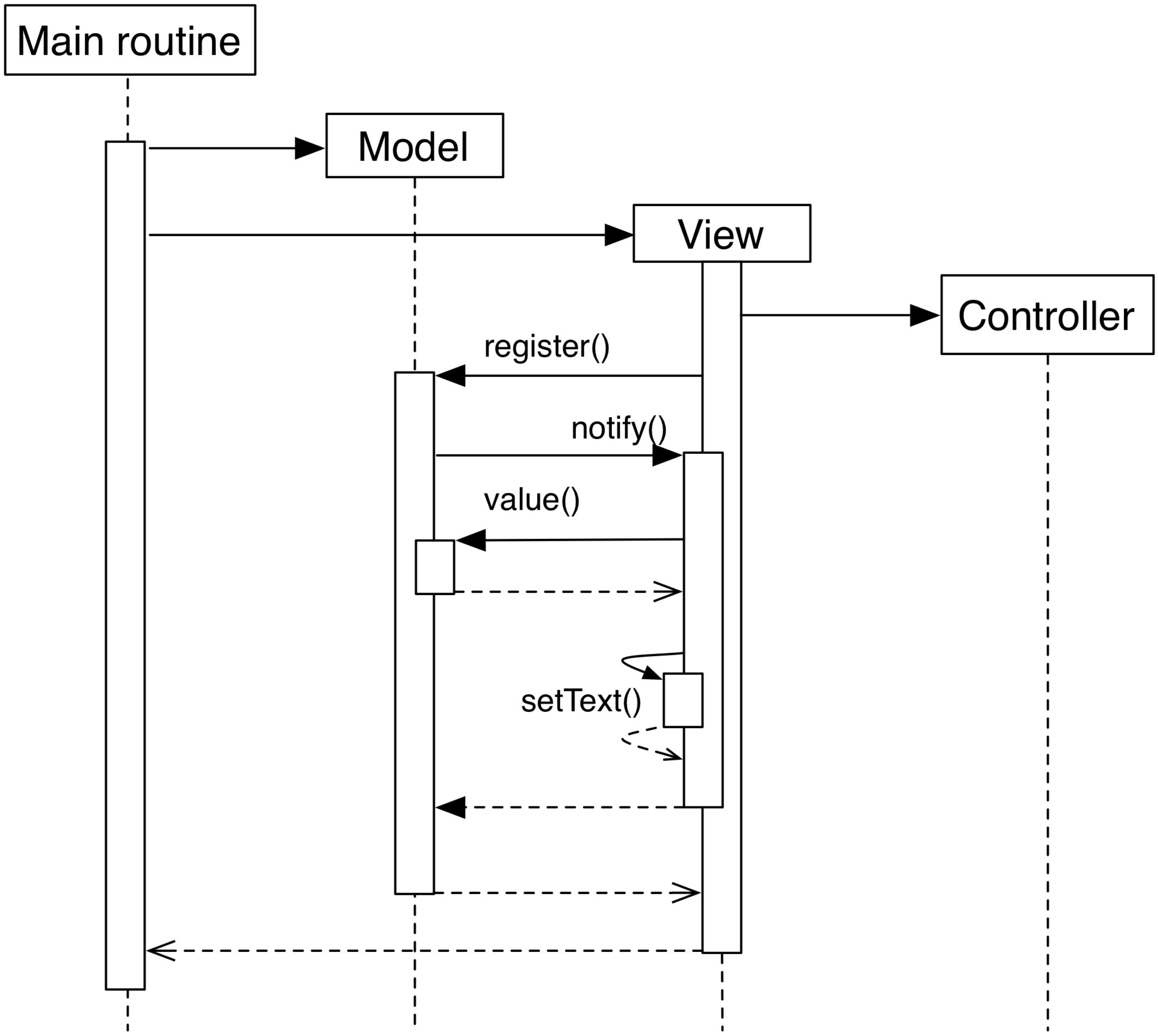
To initialize the MVC triad, the client code needs to create the Model and View, and let them be aware of each other by passing the Model to the View.

```
app = QtGui.QApplication(sys.argv)

model = Model()
view = View(model)
view.show()

app.exec_()
```

The activity diagram shows the setup code given above



This schema assumes that the controller is initialized by the View. This is generally desirable, given that View and Controller are so dependent and tailored to each other that passing the Controller from outside is not profitable.

FIXME The direct connection between View and Controller is needed for: 1) the View initializes the controller with an instance of itself at creation 2) the currently active controller can be found by traversing the view hierarchy

FIXME: Strictly speaking, the controller does not need to know about the View: the controller receives events (from the user) and modifies the model, and the model syncs against the view. In practice. the user interacts with a view object, and the associated controller handles that operation.

[1] The more knowledgeable reader may recognize that this MVC model is not the original MVC as intended in its first implementation. We will go into detail of the differences in later chapters. What is presented here is the modern reinterpretation of the original MVC, and the

one most likely to be intended when talking about "MVC". To clarify the overloaded nomenclature, I chose to refer to the original '70s design as "Reenskaug MVC", and its modern reinterpretation here presented as "Traditional MVC".

An in-depth analysis of Traditional MVC roles and components

In the previous sections we performed a progressive refactoring from Smart-UI, to Document-View, and finally to Traditional MVC. This refactoring was driven by the need for additional flexibility, separation of concerns and clarification of the different roles. MVC is, at its core, an exercise in data synchronization: the same data must be present in the GUI, the Model, and finally in any data source the Model may be using to access the data, for example a file or a SQL database. The MVC roles help us giving structure to the communication traffic needed by this synchronization ballet.

To summarize the scope of each role in Traditional MVC:

- **Model**: holds the application's state and core functionality (Domain logic).
- **View**: visually renders the Model to the User (Presentation logic).
- **Controller**: mediates User actions on the GUI to drive modifications on the Model (Application logic).

Except for the most trivial applications, multiple classes can be active in the same role and are said to belong to a specific **layer** (i.e. Model layer, View layer and Controller layer). Objects from these layers are composed into MVC Triads that give rise to the final application's behavior and aspect. This design is blessed with technical advantages:

- The clear separation of concerns between data storage, data handling, data visualization, and user interaction opens the possibility to be flexible in changing their implementation (for example, the layout of the graphical interface).

- The communication among objects is restricted on purpose and characterized by its triad interaction pattern, reducing complexity and side effects.

- Applications that need to visualize the same data in different ways, or modify them from different sources (for example, a data table and a plot) can do so while keeping the information centralized and synchronized.

- Separation of concerns leads to easier testability and thus higher reliability: each component can be tested independently from the others, with their dependencies replaced by mock objects with predictable behavior.

- Frameworks and GUI toolkits already provide MVC solutions as part of their design: you just have to “fill the blanks” to get a working application.

Additionally, MVC accelerates development, improves readability and communication of intent:

- Different teams with different skills can work in parallel on separate parts of the application: frontend developers and GUI designers can work on the visual aspect, while backend developers and storage scaling specialists can work on low-level data representation.
- By defining clear interfaces on the protagonists' classes, the code documents itself both through the API and their role within the MVC design
- MVC provides a common vocabulary to talk about roles and responsibilities in design.

A large application is composed of many different triads, each ideally decoupled from the others, except at the Model level.

The view uses the controller as a "strategy" for its handling policy with respect to user events. While technically the view could change Controller in agreement to the Strategy pattern, in practice this never happens. View and controller are tightly connected and remain so for the whole lifetime of the triad. Eventually, the controller is acted upon to modify its behavior, but it is never replaced altogether for a different one.

The Model

Entities taking the Model role are responsible for holding the running state and business functionality of the application, either as a whole or as the part that is relevant to that specific MVC Triad, as either data (stored values) or activity (routines computing relevant data). They define the protagonists of the application's domain, their mechanism of operation and cooperation. Model objects can represent, for example,

- An interface to a database, filesystem, low level driver, hardware machine
- Access to a computational backend.
- Proxies for a remote service
- A representation of business entities such as weather forecast in a specific area, people's details in a phonebook, tracks information in a music CD, student grades
- In some designs, graphical state of the GUI, such as selected items, or the X-axis scale of a plot.
- A running process.

When implemented, a Model can go from a dictionary-like of key/value pairs to a complex network of objects with well defined interfaces. Regardless of the implementation, Models in Traditional MVC must provide the following three services:

Querying: to inquire about their current state, either represented by high-level domain objects (Object Oriented approach), or through an IO layer of routines providing access to the data (Data Oriented approach). In the Object Oriented approach, the Model objects generally represents an identifiable part of the domain of your application, and provide access to data through a well-defined object-oriented interface. The Model can also perform computation, generally of information derived or associated to the main data it represents. In the Data Oriented approach, the routines “speak the domain language” and have high-level semantics to access the data, generally from a data storage (e.g. disk).

Altering: to modify the current state. The Model interface provides set methods or routines to modify its state. The Model performs consistency checks about the data it handles, enforcing fundamental integrity: for example, it can raise an exception or ignore the passed data if a method setCurrentTemperature is called passing a string instead of a float, or a method setLength is called with a negative value.

Notifying: to inform interested parties that a change has occurred in its state. The Model allows interested objects to register themselves for notifications. When a change occurs, these objects will be notified of this fact and can act accordingly, normally by synchronizing themselves against the Model's new contents.

Model objects should provide core application functionality through a clear and self-documented interface, exposing what can be done with the program's state. To operate, they can depend only on other Model objects or other components of the application that don't involve presentation, like an IO layer. The relationship among Model objects is that of a **strong dependency**.

On the other hand, a Model should not contain nor be dependent for its functionality on any graphical entity, nor contain formatting/visual logic for presentation (*e.g.* logic to make a negative value represented in red, or logic to present the date in US vs. ISO representation). Model objects should be completely unaware of how user interaction is handled by the application they live in, and should have a **weak dependency** toward its listeners via the notification generic interface.

For data modification, all the Model does is to process incoming requests in the form of method calls. Normally these requests are performed by Controllers, but a Model can also change due to requests from other subsystems (for example, a network layer), from another Model component or because it is monitoring a backend (e.g. a database, or a filesystem) and the monitored entity changes. The only entities never allowed to issue a change request to the Model are the Views.

The Model should enforce integrity of the data, but it does not necessarily enforce validity: data might be correct (for example, integers for min/max values) but overall invalid for computation (for example, if min > max). While integrity should be enforced, storing invalid data can be acceptable: depending on the application, invalid data may be marked as such in the Model by the part of the code that detects the invalidity, so that the View can represent it (for example, with a red font); An invalid state might be needed as a stepstone to reach a valid state at the end of a set of changes done by the User via the UI.

With the above guidelines and restrictions in place, the resulting implementation is robust, flexible and testable: Views and Controllers are the components that change the most as the application evolves, and a Model that is agnostic to these changes is easier to code and maintain. The Model can be tested independently from the rest of the application, and it opens itself to scripting, allowing the User to change the Model programmatically instead of through the GUI.

Not all Model objects are the same. We can generally distinguish three subclassifications of the Model layer:

```
- Service Model (Domain Model): the actual part of the Model that describes the
  business rules and objects.
- Data access: responsible for serialization and deserialization of the
  Service Model objects and persistence.
- Value Objects: data objects with particular characteristics so that
  their equivalence depends on their value, rather than their identity.
```

Keeping the application functionality fully supported by the model allows scriptability. The Model can be modified programmatically with a script that replaces the user interaction.

The View

We introduced the View as the component of our application whose role is to interact with the User, by accepting its input and showing the contents of the Model, and operates on it in **read-only**. Begin the face of the application, is also the one that is more likely to change and adapt, often under pressure of non-programming factors such as visual design and usability needs.

A View listens for Model notifications and responds by fetching and rendering the new state from the Model. This results in a strong dependency toward the Model: Views must access Model data, something that requires full dependency toward the Model's interface and existence.

Views are responsible for "purely GUI" intelligence, like handling behavior on resizing, repainting, data displaying and visual formatting. They are also in charge of handling primary events such as mouse clicks and keyboard key presses, but should not perform any modifying action on the Model as a consequence to these events. Instead, they should delegate this task to Controllers. They should also not perform any operation that is competence of the Model, nor store Model data, except for caching to improve rendering performance. Cached data are never authoritative, and should never be pushed back into the Model, or handed out to other objects.

A View is generally composed out of **Widgets**, reusable visual building blocks provided by a Widget toolkit library. Examples of widgets are buttons, checkboxes, and menus. Complex interfaces are assembled from a collection of Widgets, hierarchically contained in dialogs, windows and other visual containers. This intrinsic hierarchic nesting must be factored in when we want to go from the basic MVC given in the previous section to a real-world MVC. The hierarchy is bidirectional, meaning that containers hold references to the contained widgets, and vice versa. Widget state is normally modified from client code via method calls, having no intelligence for receiving notifications from the Model. A View adds Model observing capabilities and rendering logic to a widget or groups of widgets, either through inheritance or composition.

A view responds to Model notifications and repaint itself. This is, however, not the only reason why a View may have to inquire the Model for data and repaint itself. The view may have to do so after a show event.

MVC is not only limited to GUI representations, and Views are not necessarily graphical objects. In fact, anything that can report information to the User in some form can be classified as a View. For example, a musical notation Model can be observed by two Views:

one that shows the musical notation on screen and another that plays it on the speakers.

View can work also without a model, but of course it cannot display anything. It can accept another model. In that case, it will unregister from the previous one and register on the new one.

The Controller

The last of the components of MVC, the Controller, has the heavy duty task to make things happen by gathering, validating, and processing User events to modify the state of the application. It operates on the Model in **read-write**.

Controllers are associated to Views in a strong one-to-one mutual dependency, and can be described as the “business logic” of the View. When the View receives a primary event, it forwards execution to an appropriate Controller method, where appropriate logic modifies the state of the application. Generally, the change is applied to the Model, but depending on the problem the Controller can also directly modify the View, in particular when it changes visual state that is purely pertinent to the View and is not represented in the Model. Examples of these cases can be enabling/disabling some widget, scaling/zooming of a plot area, reordering of menu entries and so on.

A Controller generally hosts a reference to its View and the Models it modifies, and depends strongly on their interfaces and presentation semantics, at least to a degree. It may act on Models that are not the ones observed by the associated View. Like Views, a Controller can be a listener for Model notifications, when the Model state influences how the Controller interprets the User events.

the controller can act on the model, or directly on the view, if it needs to change the visual representation without involving a change in data.

specify that it's *not* View == read-only widget vs Controller == read-write widget.

Forces outdating Traditional MVC

The Traditional MVC design presented in the previous section is a modern reinterpretation of the MVC as described by Reenskaug in the 70s. The original design was developed under different constraints, and could not take advantage of the modern solutions we enjoy today.

For example, Reenskaug's Controller handled low level events, positioned the Views on the screen, and kept track of which View had focus and which Controller in the application was in charge of handling the events.

Modern environments, compared to the ones where Reenskaug MVC was developed first, have improved on a lot of boilerplate tasks: modern Views are composed of widgets provided by either a GUI Toolkit or the operating system's framework. These widgets acts both as Views and as Controllers as originally defined, because they can display and position themselves, manage focus, receive low-level events and convert them to higher level behavior: a modern LineEdit widget handles keyboard input to write text on the screen without any additional support. An application-level event loop handles events and dispatches them to the appropriate receiver.

The result of this modernization is a reduction of responsibility of the Controller, and its role has been adapted with the times. New interpretations of the old pattern emerged, and the Traditional MVC introduced earlier is an example of this adaptation. The Controller now handles high-level events from the View, rather than raw, low level events, and can eventually take the role of mutator of the Model.

On the other hand, new needs emerged from more complex and communicative GUIs, underlying toolkits, and new architectures (i.e. the web) making Traditional MVC sometimes too inflexible, sometimes too limited, and sometimes overdesigned for the specific task at hand.

In the next chapter, we will examine a palette of variations of the basic building blocks of MVC to provide development strategies for common GUI development needs.

FIXME: off-the-shelf widget sets. Reimplement widgets to define methods for events is annoying. proliferates classes. Controller was in charge of deciding when to relinquish control to other controllers. The active controller was the one handling events. Modern widgets handle damage control (e.g. due to hiding/showing) by themselves. The view is only left the task of updating against a modified model. The toolkit takes care of keeping the visual correct.

Explain widget as a UI element which has no connection to a model.

MVC Variation

Traditional MVC is excellent as a starting point for discussion, but by no means it must be considered the one and only proper way of doing MVC. In fact, we saw how Traditional MVC is outdated within the context of modern UI development. New patterns have emerged, built around fundamental MVC concepts and nomenclature, but with additional tricks, extensions and alternative design choices to satisfy modern requirements, such as the following:

- a modal dialog must allow changing values, but revert them when the Cancel button is pressed.
- a modeless dialog allows changing values while the change is visible in another window, but must be reverted if “Restore” is pressed.
- prevent typing of invalid values, for example a string in a line edit supposed to accept only digits will not accept any key presses from non-digits.
- alternatively, allow invalid entries, but disable the Ok button and mark the incorrect value red.
- and so on...

As you can see, the complexity of an application made of hundreds of menus, text input areas and buttons, plus all the possible logical dependencies among them can grow considerably. Unexpected interactions and strange communication patterns emerge in the form of bugs or application freezes. Keeping this communication network well organized and confined by enforcing a structure is of paramount importance.

Additionally you might have to fight your toolkit because it prefers a specific implementation of MVC.

Design aims at managing complexity. The MVC details given in this book are guidelines, but need to consider the actual real problem at hand. Some flexibility is needed. Strict compliance generally produces a benefit, and has better communicative consistency within the development team, but may not scale up to specific cases. In that case, reconsider the design, or relax some constraints, but aim at keeping logic encapsulated and object interaction simple and straightforward.

In this chapter we will examine alternative design in MVC able to deal with more complex use-case scenarios, constrained by requirements, architectural needs or self-documentation purposes.

Variations on the Model

- Compositing Model: Aggregate data from multiple Models, providing a single point of access to the View.
- Model Pipe: Intercept, modify, and filter data flow between Model and View.
- Application Model: Holds visual state to complement business logic state.
- Side-by-Side application model: Holds complementary information to the current model.
- UI Retrieving Model: Model retrieves information from the User through UI elements.
- ModelController: Aggregates Model and Controller logic in a single object.
- Local Model: Preserve the original Model state so that changes can be reverted.
- Value Model: Trivialize complex Model interface to a simple, universal interface.
- Proxy Model: Acts as proxy, hiding complexity to access resources from a backend.
- Collection Model: Holds and aggregates Model objects of the same type.
- View-aware Model: A model that knows its views interface beyond simple notification delivery.
- Caching Model: Holds data in local cache to mitigate slow access.
- Data/Metadata Model: Separate access to slow and bulky data vs. fast, lean and descriptive.
- Recording Model: Record the changes for later consumption.
- Transactional Setting: Setting multiple attributes at once with late notification.

Compositing Model

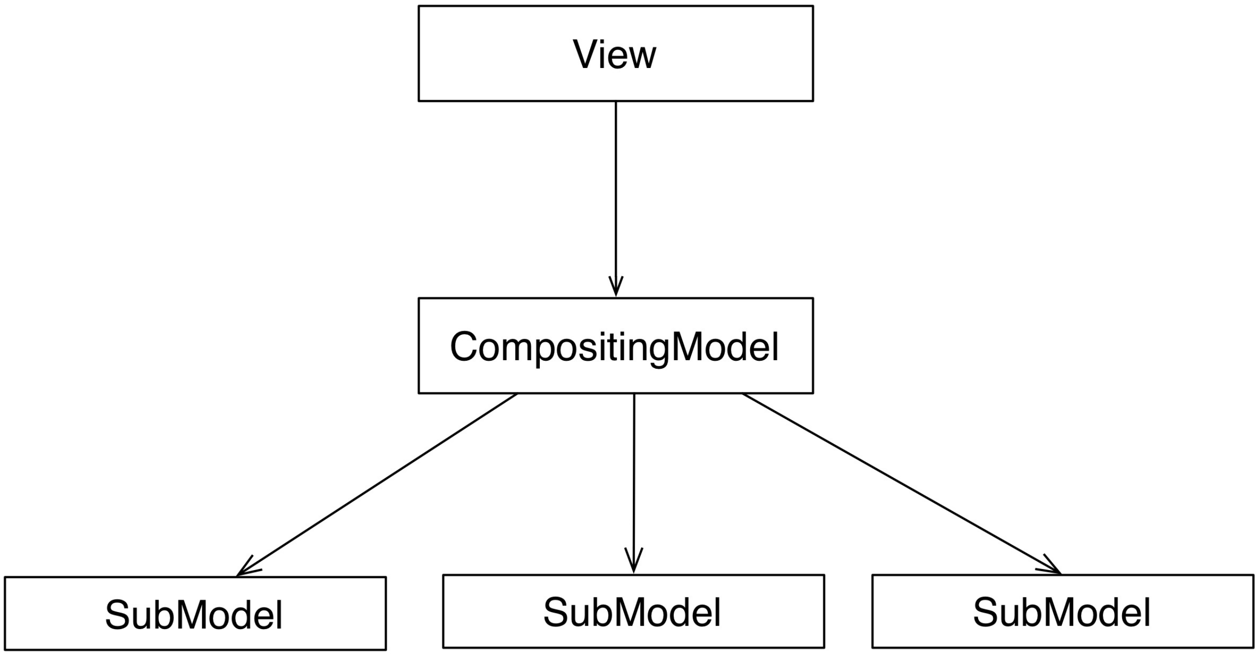
Motivation

A Compositing Model aggregates data from multiple Model objects so that the View has a single and uniform point of access for its data source. Two different sub-types of a Compositing Model exist, each addressing a different use cases:

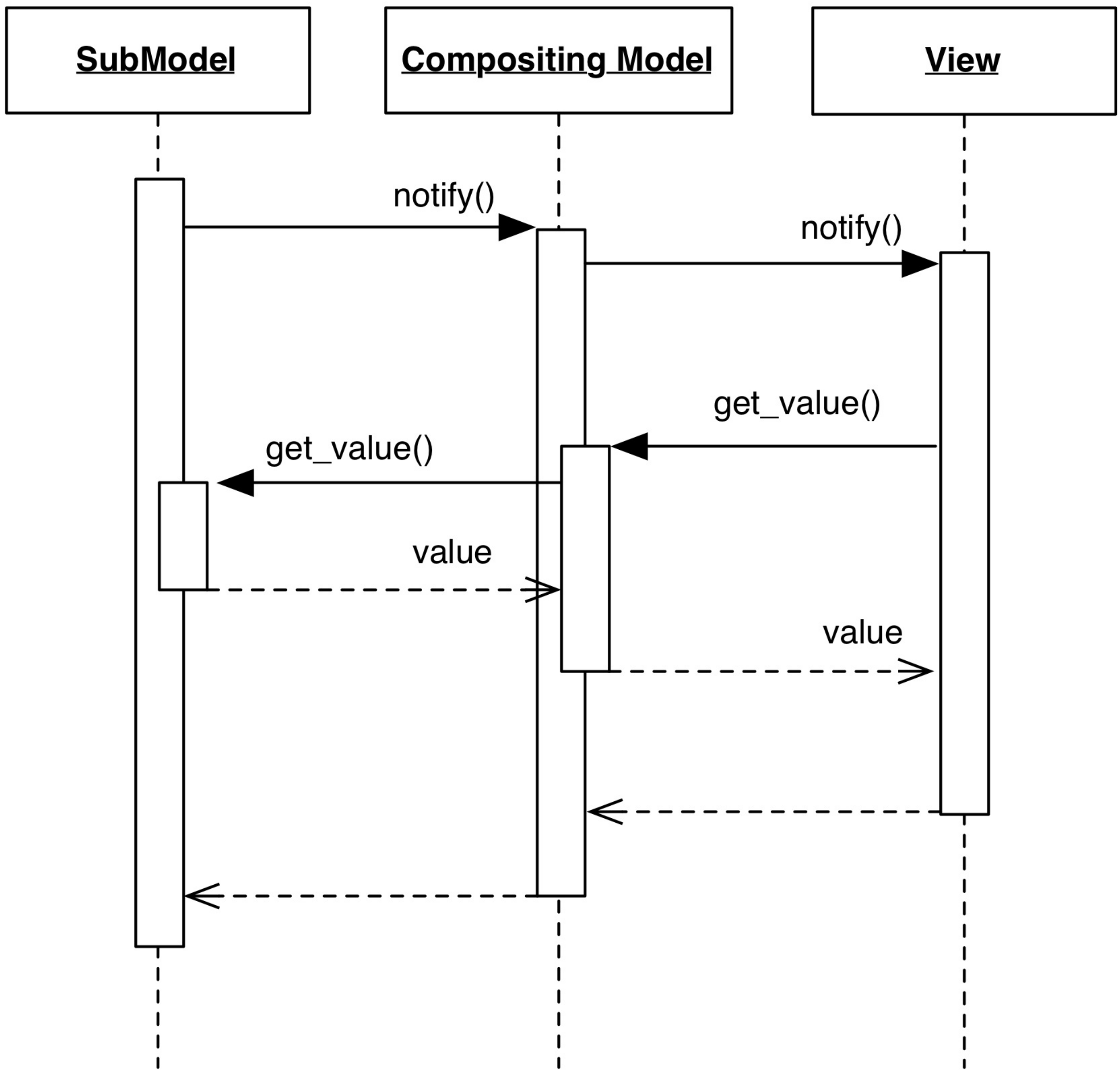
- A **Union Compositing Model** performs union of homogeneous data originating from different sources. For example, a Union Model can present data from multiple files, each handled by a SubModel.
- A **Join Compositing Model** extracts and combines relevant information from heterogeneous Models and provides a convenient interface to the result. For example, a `CustomerHistory` Model could combine Models `Customers` and `Orders`, returning the combined information to the View.

Design

The Compositing Model acts both as listener and notifier. It holds references to its SubModels, and registers onto them as a listener. Its life cycle can be temporary and disjoint from the life cycle of its SubModels.



Notifications from individual SubModels are received and re-issued by the Compositing Model to notify the View. Vice-versa, data requests issued by the View on the Compositing Model are routed to the appropriate SubModel.



By its very nature, the Compositing Model is expected to be the only endpoint for a specific View, at least for data retrieval. The Controller, on the other hand, can either:

- issue change requests on the Compositing Model, which in turns forwards the change request to the appropriate SubModel according to some criteria. The Compositing Model is therefore acting as a surrogate Controller
- issue change requests directly on any of the Submodels. The SubModel will then issue the notification, which is forwarded to the View by the Compositing Model.

It is worth pointing out that nothing prevents other Views (or Controllers) to interact directly with any of the SubModels.

The above design considerations are common between the Union and the Join Compositing Models. The most substantial difference between them is in their interface. Union Models generally have the same interface as their SubModels, and Views can display either of them

transparently. Join Models, on the other hand, likely provide a different interface to derived data, and the View is designed to target only this interface.

Practical example

A practical example of the Union Compositing Model for an AddressBook application is here presented. The `AddressBook` class aggregates a list of SubModels, each extracting data from a different file source:

```
csv1_model = AddressBookCSV("file1.csv")
xml_model = AddressBookXML("file.xml")
csv2_model = AddressBookCSV("file2.csv")

address_book = AddressBook([csv1_model, xml_model, csv2_model])
```

A naive implementation for `AddressBookCSV` is here shown to illustrate its interface. The common base class `Model` provides notification services by implementing `register` , `unregister` , `notify_listeners` , and the listeners set

```
class AddressBookCSV(Model):
    def __init__(self, filename):
        super(AddressBookCSV, self).__init__()
        self._filename = filename

    def num_entries(self):
        try:
            return len(open(self._filename, "r").readlines())
        except:
            return 0

    def get_entry(self, entry_number):
        try:
            with open(self._filename, "r") as f:
                line = f.readlines()[entry_number]
                name, phone = line.split(',')
                return { 'name' : name.strip(), 'phone' : phone.strip()}
        except:
            raise IndexError("Invalid entry %d" % entry_number)

    def append_entry(self, name, phone):
        with open(self._filename, "a") as f:
            f.write('{},{}\n'.format(name, phone))

        self.notify_listeners()
```

The `AddressBook` class is a Union Compositing Model implementing the same interface of the SubModels

```
class AddressBook(Model):
    def __init__(self, sub_models):
        super(AddressBook, self).__init__()

        self._sub_models = sub_models

        for m in self._sub_models:
            m.register(self)

    def num_entries(self):
        return sum([m.num_entries() for m in self._sub_models])

    def get_entry(self, entry_number):

        accumulated = itertools.accumulate(
            [m.num_entries() for m in self._sub_models])

        source_idx = [x <= entry_number for x in accumulated].index(False)

        return self._sub_models[source_idx].get_entry(
                               entry_number - accumulated[source_idx])

    def append_entry(self, name, phone):
        self._sub_models[-1].append_entry(name, phone)

    def notify(self):
        self.notify_listeners()
```

The class accepts an arbitrary number of SubModels at initialization, and registers as a listener on each of them. It implements the same interface, retrieving data from the SubModels: the number of entries is the sum of the SubModel entries, and the `get_entry` method returns the entry from the appropriate SubModel. `append_entry` is used to add a new entry to the models. In this case, the new entry is added to the last SubModel. Note how the issuing of the notification is left to the SubModel. The notification from the SubModel is then forwarded by the Compositing Model it to its listeners.

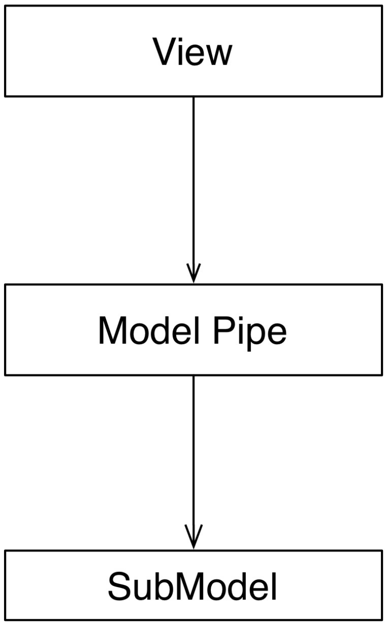
Model Pipe

Motivation

The **Model Pipe** is a variation similar in concept to a UNIX pipe: it intercepts and manipulates the data flowing from Model to View. Like in a UNIX pipe, Model Pipes with different manipulation logic can be chained together to perform sequential alteration of data. Typical application of Model Pipe are for filtering and sorting.

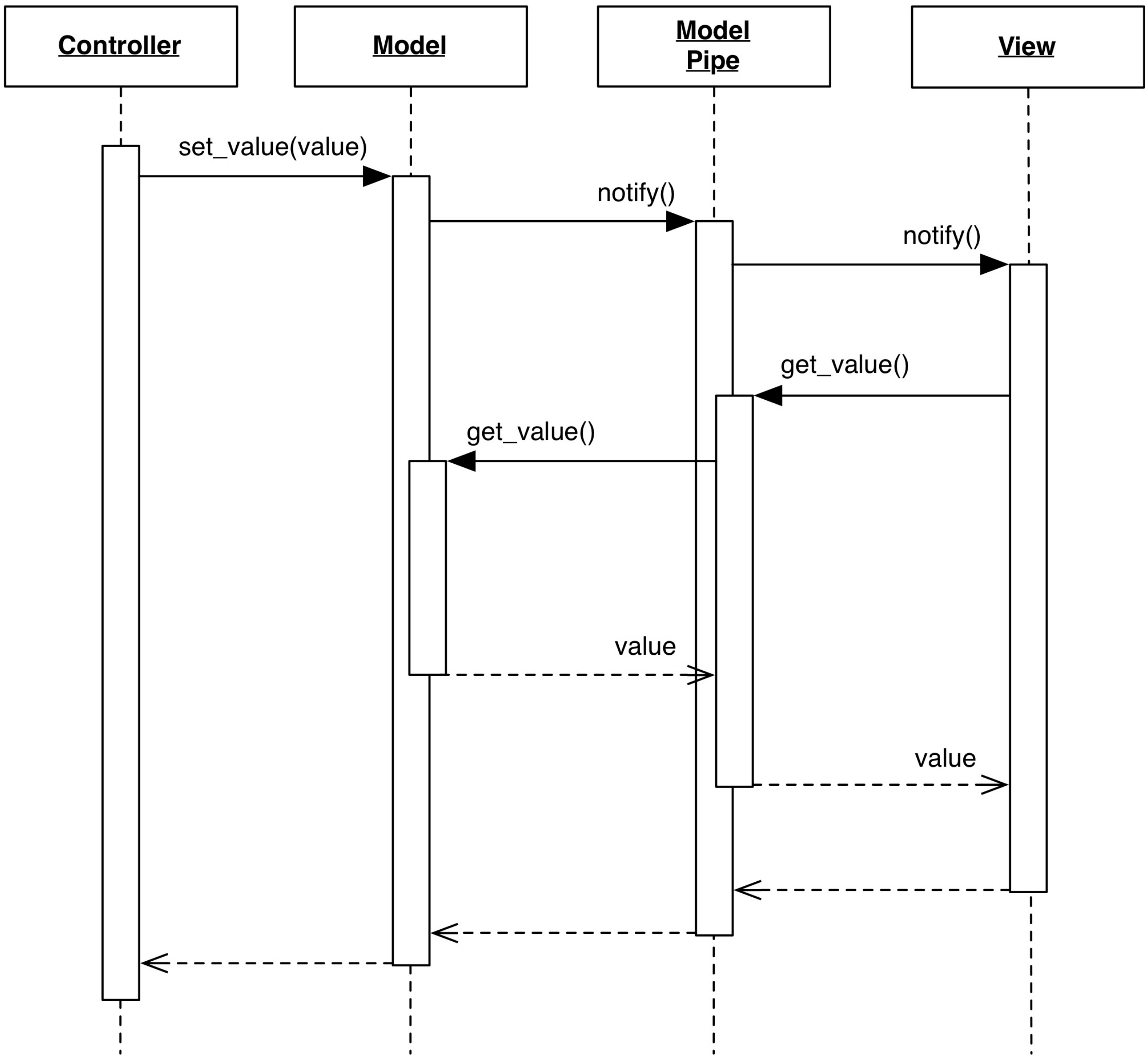
Design

The Model Pipe encapsulates data transformation logic in a dedicated, reusable class. Different Pipe classes can be created, each with specific capabilities.



To be compatible with the View, a Model Pipe implements the same interface of the SubModel, eventually extending it to provide access to additional state it might contain. For example, a filtering Model Pipe will host data about the current filter string. Controllers acting on the Model Pipe act on this state.

The Model Pipe is a listener of the SubModel and forwards its events. In addition, it performs notification when its internal state changes in a way that modifies the resulting data.



while the manipulation of the SubModel's data is performed directly on the SubModel itself.

Practical Example: Qt QSortFilterProxyModel

Qt provides a Pipe Model with sorting and filtering functionality: `QSortFilterProxyModel`. It is worth noting that Qt calls this Model "Proxy Model", but in the context of this book, Proxy Model refers to a different Model design.

The following code sets up a Pipe Model connection between `MyModel` (our implementation of `QAbstractItemModel`) and a Tree View. The Tree View observes the Pipe Model, which in turns uses `MyModel` as a data source.

```
tree_view = QTreeView()
model = MyModel()
pipe_model = QSortFilterProxyModel()
pipe_model.setSourceModel(model)
tree_view.setModel(pipe_model)
```

The Model Pipe provides an appropriate interface to configure the filter, such as `setFilterRegExp`, and mapping routines `mapToSource()` and `mapFromSource()` to convert between index of an Item in the Model and index of that Item in the View.

Note how the `tree_view.setModel()` accepts a `QAbstractItemModel`, which is implemented by both `MyModel` and by `QSortFilterProxyModel`. As explained earlier, this design allows the View to accept either Model, and remain unaware of the existence of the pipeline.

Application Model (MMVC)

Motivation

In Traditional MVC we pointed out that a Model object should not contain GUI state. In practice, some applications need to preserve and manage state that is only relevant for visualization. Traditional MVC has no place for it, but we can satisfy this need with a specialized Compositing Model: the **Application Model**, also known as Presentation Model. Its submodel, called **Domain Model**, will be kept unaware of such state.

An Application Model is closer to the View than a Domain Model, and therefore able to take into account specific needs of the View it is addressing: in a scrollable area, where only a part of the overall Model is visible it can hold information about the currently visible portion of the Domain Model, and suppress those notifications reporting changes in data currently not visible, preventing a useless refresh. It can also be used to distill information from multiple Domain Models, producing something that is relevant for its View. For example, our Domain Model may be made of objects representing the employees in a company, company departments and so on, in a rather elaborate network. If the View wants to display a list of employees regardless of the department, maybe with a checkbox to select them for further processing, it is convenient to have an Application Model presenting data to the View as a list, gathering the details from the Domain Model objects (non-graphical information) while at the same time keeping track and presenting the checkbox state as well (graphical information). As a drawback, it is much less reusable: multiple Views can interact with the same Application Model only if they agree on the visual state representation (e.g. we want both the Dial and the Slider red when over the rpm limit).

Some implementations of Application Model push its responsibilities even further than purely GUI state: it is, quite literally, the model of the application, and it is responsible for modifying application state directly on the application itself. For example, it might enable/disable menus, show or hide widgets, validation of the events. Most of the visual logic will be responsibility of this model object, rather than the controllers. This interpretation has deep implications for the Dolphin Model View Presenter, which will be examined later.

FIXME: Application model represents the GUI state without the GUI. it contains the logic for enabling/disabling checkboxes, for example. FIXME: Application model can contain selection.

FIXME: Some logic may not be possible to extract from the View and put into the presentation model, especially if this logic is deeply rooted in the graphical characteristics of the visual state. Examples are options that depends on the screen resolution, or the visual

positioning of the mouse within the window.

Design

Practical Example

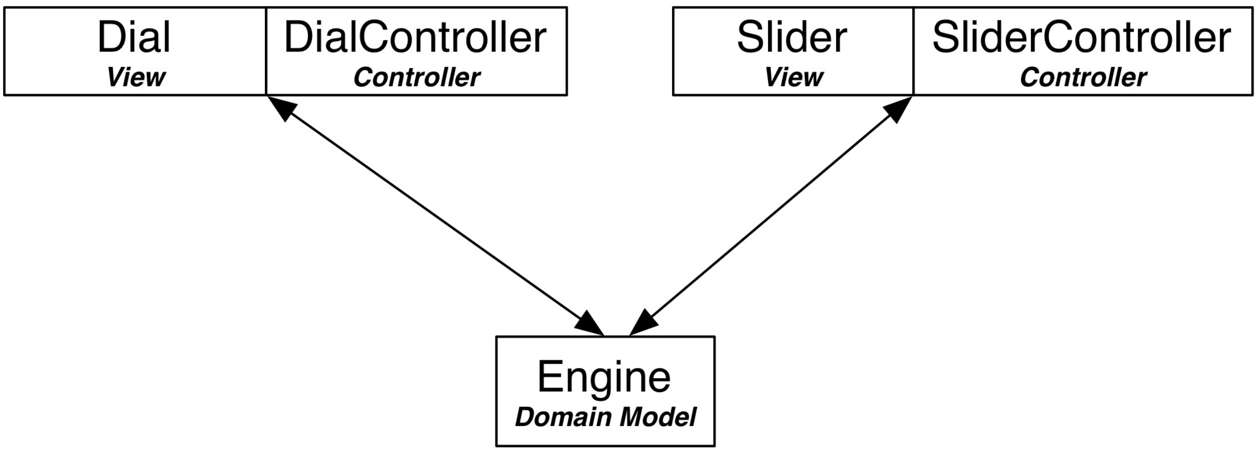
To present a practical example. imagine having a Domain Model representing an engine

```
class Engine(BaseModel):
   def __init__(self):
       super(Engine, self).__init__()
       self._rpm = 0

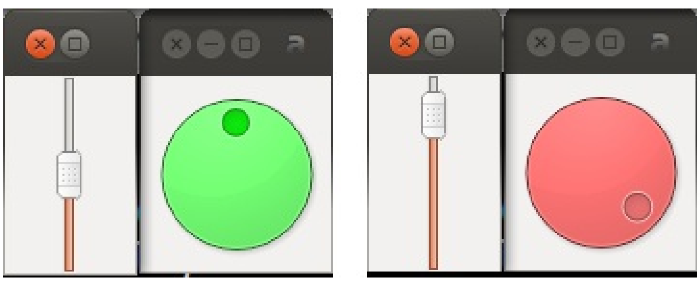
   def setRpm(self, rpm):
       if rpm != self._rpm:
           self._rpm = rpm
           self._notifyListeners()

   def rpm(self):
       return self._rpm
```

Initial specifications require to control the revolution per minute (rpm) value through two Views: a Slider and a Dial. Two View/Controller pairs observe and act on a single Model



Suppose an additional requirement is added to this simple application: the Dial should be colored red for potentially damaging rpm values above 8000 rpm, and green otherwise.



We could violate Traditional MVC and add visual information to the Model, specifically the color

```
class Engine(BaseModel):
  # <proper adaptations to init method>

  def dialColor(self):
     if self._rpm > 8000:
        return Qt.red
     else:
        return Qt.green
```

With this setup, when the Dial receives a change notification, it can inquire for both the rpm value to adjust its position and for the color to paint itself appropriately. However, the Slider has no interest in this information and now the Engine object is carrying a Qt object, gaining a dependency against GUI. This reduces reuse of the Model in a non-GUI application. The underlying problem is that the Engine is deviating from business nature, and now has to deal with visual nature, something it should not be concerned about. Additionally, this approach is unfeasible if the Model object cannot be modified.

An alternative solution is to let the Dial View decide the color when notified, like this

```
class Dial(View):
   def notify(self):
       self.setValue(self._model.rpm())
       palette = QtGui.Qpalette()

       color = Qt.green
       if self._model.rpm() > 8000:
           color = Qt.red

       palette.setColor(QtGui.Qpalette.Button, color)
       self.setPalette(palette)
```

Once again, this solution is impractical, and for a complementary reason: the View has to know what is a dangerous rpm amount, a business-related concern that should be in the Model. This solution may be acceptable for those limited cases when the logic connecting the value and its visual representation is simple, and the View is designed to be agnostic of the meaning of what is showing to the User. For example, a label displaying negative values in red may be used to show bank account balances. The real meaning of a negative balance, the account is overdrawn, is ignored. A better solution would be to have the BankAccount Model object provide this logic as isOverdrawn(), and the label color should honor this semantic, not the one implied by the numerical value.

Given the point above, it is clear that the Engine object is the only entity that can know what rpm value is too high. It has to provide this information, leaving its visual representation strategy to the View. A better design provides a query method `isOverRpmLimit`

```
class Engine(BaseModel):
   <...>
   def isOverRpmLimit(self):
       return self._rpm > 8000
```

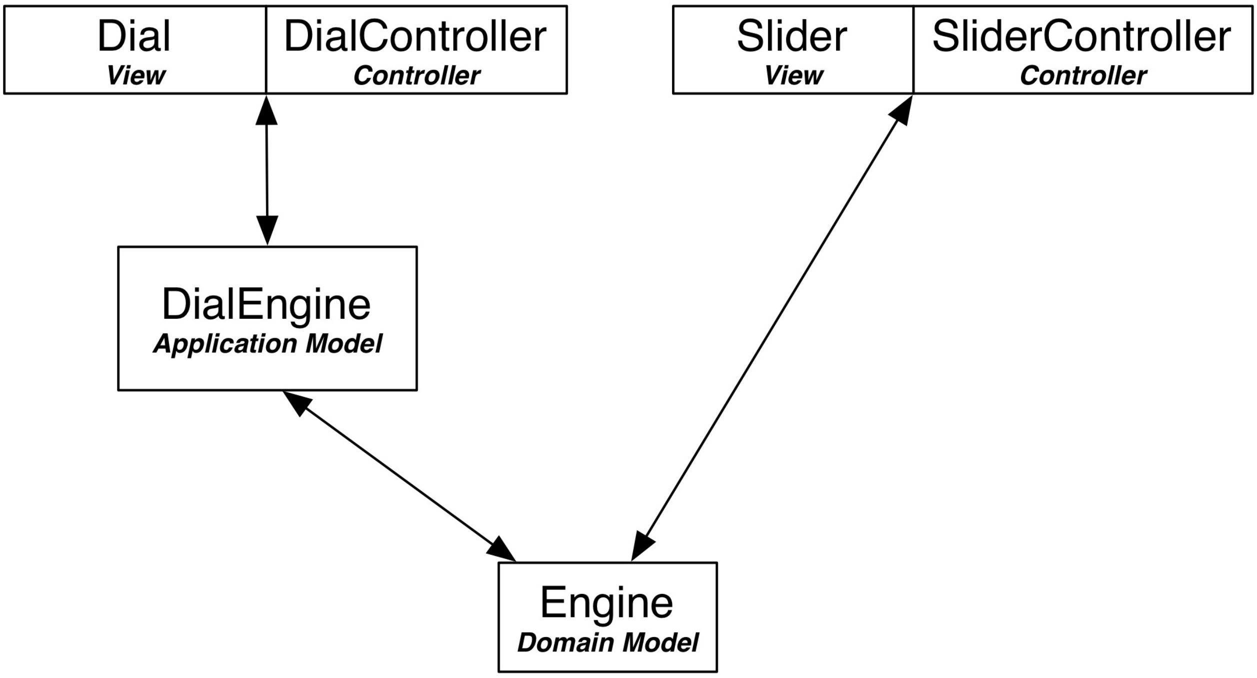
The View can now query the Model for the information and render it appropriately

```
class Dial(View):
   def notify(self):
       <...>
       color = Qt.red if self._model.isOverRpmLimit() else Qt.green

       palette.setColor(QtGui.QPalette.Button, color)
       self.setPalette(palette)
```

This solution respects the semantic level of the business object, and allows to keep the knowledge about excessive rpm values in the proper place. It is an acceptable solution for simple state.

With this implementation in place we can now extract logic and state from Dial View into the Application Model DialEngine. The resulting design is known as Model-Model-View-Controller



The DialEngine will handle state about the Dial color, while delegating the rpm value to the Domain Model. View and Controller will interact with the Application Model and listen to its notifications. Our Application Model will be implemented as follows. In the initializer, we register for notifications on the Domain Model, and initialize the color

```
class DialEngine(BaseModel):
 def __init__(self, engine):
   super(DialEngine, self).__init__()
   self._dial_color = Qt.green
   self._engine = engine
   self._engine.register(self)
```

The accessor method for the color just returns the current value

```
class DialEngine(BaseModel):
    # ...
    def dialColor(self):
        return self._dial_color
```

The two accessors for the rpm value trivially delegate to the Domain Model

```
class DialEngine(BaseModel):
    # ...
    def setRpm(self, rpm):
        self._engine.setRpm(rpm)

    def rpm(self):
        return self._engine.rpm()
```

When the `DialController` issues a change to the Application Model through the above accessor methods, this request will be forwarded and will generate a change notification. Both the Slider and the Application Model will receive this notification on their method notify. The Slider will change its position, and the Application Model will change its color and reissue a change notification

```
class DialEngine(BaseModel):
    # ...
    def notify(self):
        if self._engine.isOverRpmLimit():
          self._dial_color = Qt.red
        else:
          self._dial_color = Qt.green

        self._notifyListeners()
```

The DialView will handle this notification, query the Application Model (both the rpm value and the color) and repaint itself. Note that changing the `self._dial_color` in `DialEngine.setRpm`, as in

```
class DialEngine(BaseModel):
    # ...
    def setRpm(self, rpm):
        self._engine.setRpm(rpm)

        if self._engine.isOverRpmLimit():
            self._dial_color = Qt.red
        else:
            self._dial_color = Qt.green
```

instead of using the `notify` solution given before, would introduce the following problems:

- the dial color would not change as a consequence of external changes on the Domain Model (in our case, by the Slider)
- There is no guarantee that issuing `self._engine.setRpm()` will trigger a notification from the Domain Model, because the value might be the same. On the other hand, the Application Model might potentially change (although probably not in this example), and should trigger a notification to the listeners. Solving this problem by adding a self._notifyListeners call to DialEngine.setRpm will end up producing two notifications when the Domain Model does issue a notification.

FIXME In practice, application model is a UI model. FIXME VisualWorks. FIXME application model may need to talk to the view or the controller directly, instead of notification.

Side-by-Side Application Model - Selection Model

Motivation

An alternative approach to the previously described Application Model is to keep View state in a separate Model that is not wrapping the Domain Model, but is instead complementing it as an additional Model. The View depends and listens to both Models. This setup is commonly encountered when dealing with selection: the Domain Model may contain a list of items, with the View showing this list as selectable entities. Changing the selection will not modify the Domain Model, but will instead act on the Selection Model by including or removing items from the Selection Model.

Having a View dependent on two Models generally introduces no issues, provided that the two Models represent independent data. In the case of Selection Model, it is not the case, and particular care must be taken to handle proper synchronization and event handling. When the Domain Model changes, these changes can influence the validity of the Selection Model. For this reason, Model-to-Model communication from the Domain to the Selection Model to keep this validity.

Consider as an example the following scenario: a Domain model contains three items A, B, and C. The Selection Model might contain ordinal selection for the second element, that is, B. If however a new element A2 is inserted between A and B, the selection must be updated to refer to the third element. If selection was based on a unique identifier of the item, instead of a numeric position, the synchronization would not be needed. However, a similar problem would happen if item B is removed. The selection would now refer to an unexistent item, and must be removed. These scenarios require extensive discussion that will be presented later in this section.

The Selection Model can also be observed or modified by other Views or Controllers, granting a relevant amount of flexibility for complex selection scenarios. For example, a controller associated to a menu action may want to select all elements whose name matches a regular expression.

[picture]

Obviously, the Application Model keeps registering itself on the Domain model

```
class DialViewModel(BaseModel):
  def __init__(self, engine):
  super(DialViewModel, self).__init__()
     self._dial_color = Qt.green
     self._engine = engine
     self._engine.register(self)

  def color(self):
     return self._dial_color

  def notify(self):
     if self._engine.isOverRpmLimit():
        self._dial_color = Qt.red
     else:
        self._dial_color = Qt.green
     self._notifyListeners()
```

The dial now registers to both Models, and listens to notifications from both.

```
class Dial(QtGui.QDial):
  # <....>
  def setModels(self, model, view_model):
     if self._model:
        self._model.unregister(self)
     if self._view_model:
        self._view_model.unregister(self)

     self._model = model
     self._view_model = view_model

     self._controller.setModel(model)
     self._model.register(self)
     self._view_model.register(self)

  def notify(self):
     self.setValue(self._model.rpm())
     palette = QtGui.QPalette()
     palette.setColor(QtGui.Qpalette.Button,self._view_model.color())
     self.setPalette(palette)
```

Note how the Dial cannot differentiate which of the two Models is delivering the message, and how in particular it will be potentially notified twice: once by the change in the Domain model, and another time by the change in the Application Model, in itself triggered by the previous change in the Domain model. Particular care may be needed if the notify method is time consuming. Another case of Application Model usage is a plot with changing scale. The state of the View (its scale and positioning) is part of a “separate model” that is pertinent only to the View. The Domain model, which holds the plot data, should not be involved in the zoom factor or plot limits.

A side-by-side solution is frequently used to implement selection, a common GUI paradigm to operate on a data subset. Selected data normally have a different visual aspect, such as highlighting or a checkbox. This information is then used to drive operations on the specified subset. Selection has therefore a dualistic nature of holding state that is both visual and business related. A trivial strategy is to include selection state directly on the Domain Model, for example as a flag associated to the item. Depending on the application, this may or may not be an appropriate solution: if two Views observe the same Model, and an item is selected in one View, you might or might not want the other View to obtain this selection information. For example, a GUI allowing the user to select elements from a list, but also have a label saying “3 items selected” would work with selection on the Domain Model. If selection cannot be shared between Views, or we want to keep selection as an independent concern, a sensible strategy is to host it as a separate side-by-side Selection Model.

One problem with a Selection Model is that it must be tolerant to changes in the Domain Model. If a selected entity is removed from the Domain Model, the selection status must be cleared of that entity. This is important, because if the Selection Model is then used to perform collective operations (for example, change the color of all selected items) an operation will be attempted on an item no-longer existing in the Domain Model. Add operations are also not immune from problems: the Selection Model might have to resize itself to match the Domain Model, so that it does not go out of bounds when inquire is performed about the selection status of the new entries. Modifications may reorder and invalidate indexes in the Domain Model, making the selection outdated. Finally, when synchronization is achieved between the Domain Model and the Selection Model, the View will be notified twice: once by the change in the Domain Model, and again by the Selection Model.

invert selection, complex selections, select all, select none. If data is added, removed, or modified in the model, the Selection Model must respond accordingly. For example,

XXX : point out the importance of keeping the two states separate in the notification and the handling of the two notifications should not overlap. Otherwise you will have double execution of the same code which may be time consuming. Also note how the handling of the second notification must consider the possibility that the two notifications are in an invalid order, and that could introduce unexpected content matching in the View (e.g. model notifies view then selection vs. model notifies selection then view)

The selection model acts both as a View (for the other selectable model) and as a Model (for the view).

When the selection receives notification of change from its model, it needs to check if it's still consistent against it. If not, it needs to recover by invalidating its state.

needs model to model communication to keep consistency, which becomes hard to maintain.

Point out what number of rpm is too high is domain logic, and what color it should be is view logic.

UI Retrieving Model

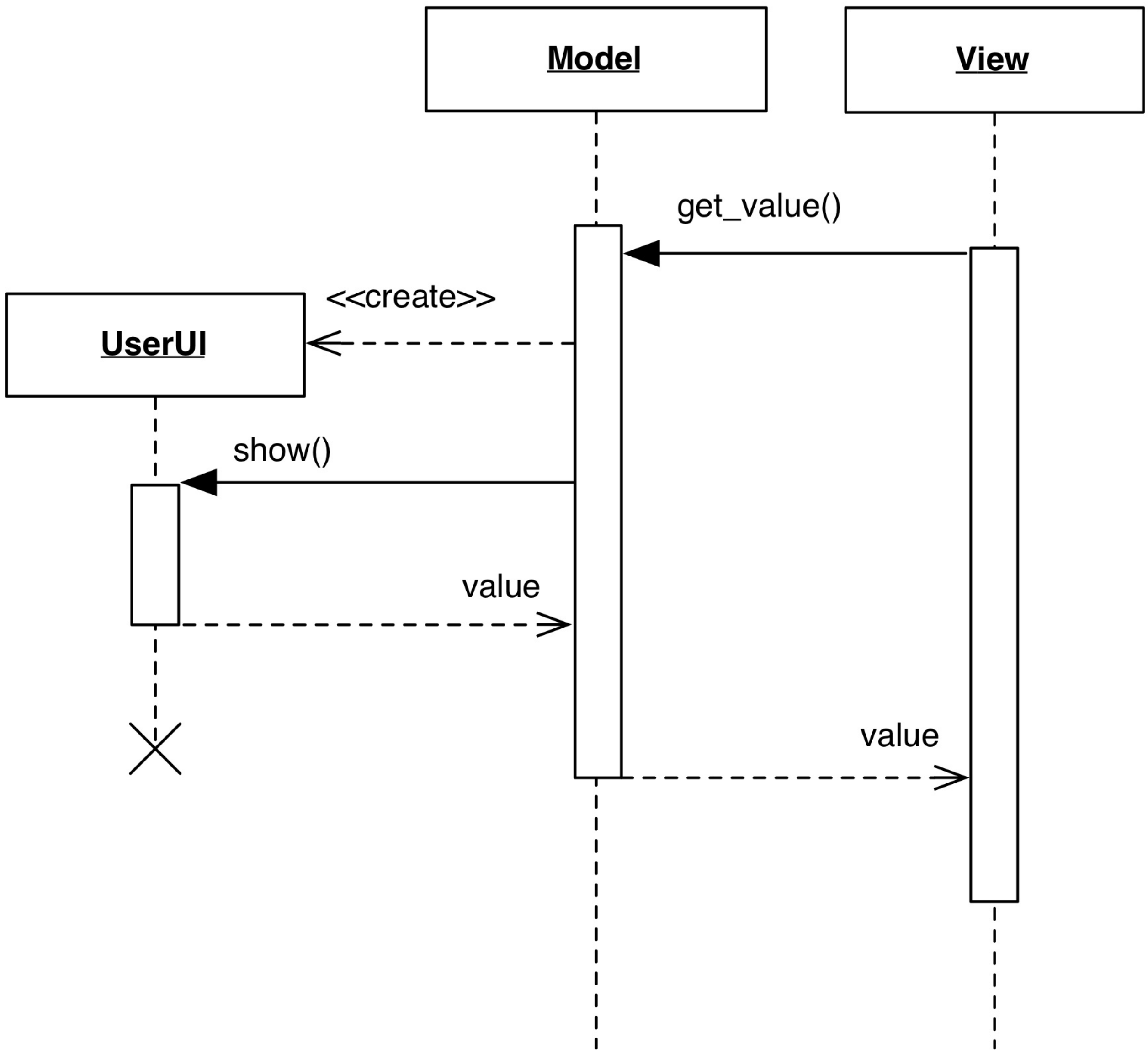
Motivation

An important constraint about the Model that we stressed throughout the book is that it should never be involved in GUI-related business. **UI Retrieving Model** breaks this rule.

Among a Model's task is the need to retrieve information from a data source, such as a file, memory, or database. In some cases, the Model may have to retrieve information that only a direct session with the User can provide. We can break the rules and let the Model create a UI dialog to inquire the User for this information. In other words, the Model consider the User as a database, whose API involves the use of graphical widgets.

Design

The interaction diagram design of a UI retrieving Model is depicted in Figure



The core of the design is the short-lived `UserUI` object. In its most basic form this object is a modal dialog, prompted to the User and awaiting for input. The modal nature of the dialog guarantees a synchronous handling of the operation.

Once the dialog is closed, the value is gathered and returned to the requesting client. Depending on the nature of the retrieved information, it might be useful for the Model to cache it instead of prompting the User again.

Practical example

One trivial example of this pattern would be a Model object representing the User session, with a password field initially set to `None`. When the program starts the session, it will ask for the credentials to the Model layer. Asking the username will return the proper string, but asking the password may return None if no password has been set yet. The Model is then authorized to retrieve this information from the User by a GUI dialog, maybe trying some non-interactive strategies first, like checking in a configuration file.

```
class Model(object):
    def get_password(self):
        if self._password is None:
            password, ok = QtGui.QInputDialog.getText(
                None,
                "Please input your password",
                "Password:",
                QtGui.QLineEdit.Password)


            if ok:
                self._password = password

        return self._password

    def clear_password(self):
        self._password = None
```

When the user acts on the dialog, the Model stores the password and returns it to the Presenter layer, which then proceeds with the authentication logic if the result is a string. If the password is found to be incorrect, the Controller will ask the Model to `clear_password()`, and the user will be prompted again.

ModelController

Motivation

In some implementations, Model state may be small and only relevant for a specific View, making the split between Model and Controller classes excessive. Merging these classes leads to a **ModelController**, which can be seen as a "Model with GUI intelligence", or as a "Controller with state".

Compared to a traditional MVC solution, ModelController has the following disadvantages:

- As stated in the opening paragraph, it complicates handling of multiple Views
- Potentially carries a dependency against the UI layer if, for example, needs to interpret pure UI events. This complicates its reuse outside of a UI application
- It is generally less flexible, less reusable, and more complex to test for the reason give above.

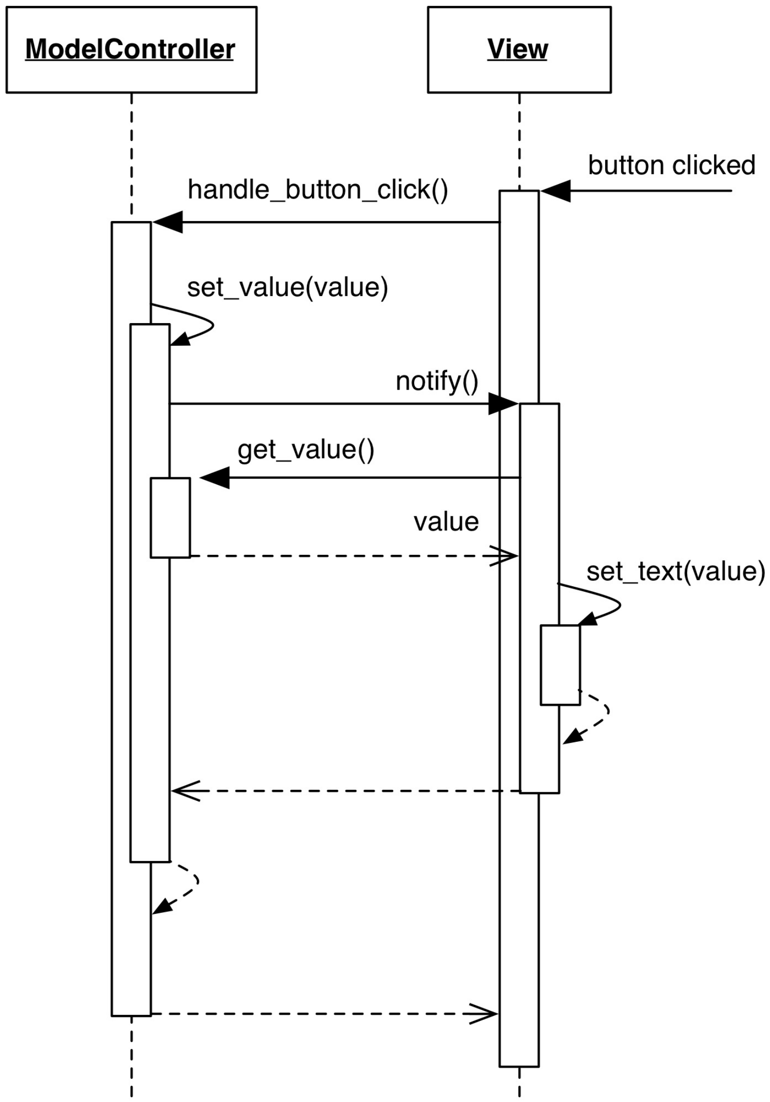
The motivation for the first point is twofold: first, the Controller part would have to differentiate which View is sending an event. Second, different Views may have a different visual implementation and thus generate different types of events. The ModelController interface would have to accommodate nomenclature and behavior of all Views, leading to a bloated and complex interface.

A ModelController approach is not necessarily wrong, but tends to become brittle and carry excessive responsibility. It is therefore limited to straightforward cases.

Design

ModelController hosts the logic to manipulate its internal state in response to UI events, applying both consistency and business logic, while at the same time handling events from the View:

- The View dispatches UI events to the ModelController
- The ModelController alters its state and notifies the View of changes
- The View requests the new state from the ModelController and updates itself.



Practical example

Trivially, a ModelController class holds state, instead of delegating it to a Model object

```
class ModelController(Model):
    def __init__(self):
        self._value = 0

    def button_pressed(self):
        self._value += 1
        self.notify_listeners()
```

This pattern can be observed, for example, in Enthought Traits

```
from traits.api import *
from traitsui.api import *

class ModelController(HasStrictTraits):
    value = Int(0)
    increment_btn = Button()

    traits_view = View(
        Item("value"),
        UItem("increment_btn", label="Increment"),
        )

    def _increment_btn_fired(self):
        self.value += 1


mc = ModelController()
mc.configure_traits()
```

The above code, thanks to the Traits/TraitsUI framework, produces a window containing an editable text field (showing the value zero) and a PushButton. Every time the button is pressed, the `_increment_btn_fired` handler is called. By design of the class and the framework, the class is a ModelController because it holds both Model state (`value`) and Controller duties (`_increment_btn_fired`). Although invisible, validation logic is also performed in the ModelController by the Traits framework, which rejects values in the text field not conforming to an integer.

Value Model

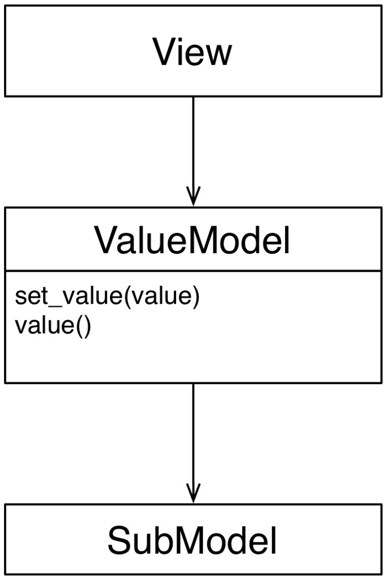
Motivation

A Value Model is an Adapter class extracting a single value from an adaptee (SubModel). The extracted value is exposed thought a uniform and trivial interface: a getter `ValueModel.value()` , a setter `ValueModel.set_value()` , and notification.

Views and Controllers using a Value Model can be extremely simple and off-the-shelf, provided that they can handle the Value Model interface. They can ignore the nature of the adapted SubModel, leaving to the Value Model the responsibility of taking care of the details.

Design

The Value Model class acts as an adapter



A trivial implementation of a Value Model would be:

```
class ValueModel(Model):
    def __init__(self, sub_model):
        self._sub_model = sub_model

    def set_value(self, value):
        # do potentially complex logic on self._sub_model
        # to appropriately manipulate the passed value
        # This method triggers a self.value_changed notification.

    def value(self):
        # do potentially complex logic on self._sub_model
        # to extract a single value
```

Many different Value Model classes can be implemented, each one adapting a different SubModel, or operating over different parts of a SubModel. Views and Controllers interact with the Value Models through the minimalist interface, and are therefore agnostic of the Value Model used.

Practical Example

One could adapt a `Customer` object through two Value Models: `NameValueModel` and `SurnameValueModel`.

```
class NameValueModel(Model):
    def __init__(self, customer):
        self._customer = customer

    def set_value(self, value):
        self._customer.name = value
        self.notify_observers()

    def value(self):
        return self._customer.name

class SurnameValueModel(Model):
    def __init__(self, customer):
        self._customer = customer

    def set_value(self, value):
        self._customer.surname = value
        self.notify_observers()

    def value(self):
        return self._customer.surname
```

Each of these two ValueModels can use an off-the-shelf `StringWidget` View, agnostic of the actual nature of the `Customer` model and retrieving/modifying data through the Value Model interface.

Proxy Model

Note: In the context of Qt MVC, a Proxy Model is a Model Pipe design. Django also defines a Proxy Model concept, but it is unrelated to the one expressed here: the application of the Proxy design pattern to a Model object.

Motivation

We want to access an external resource, such as one provided by an HTTP server or a database. Proxy Model encapsulate the logic needed to access the remote service, specifically:

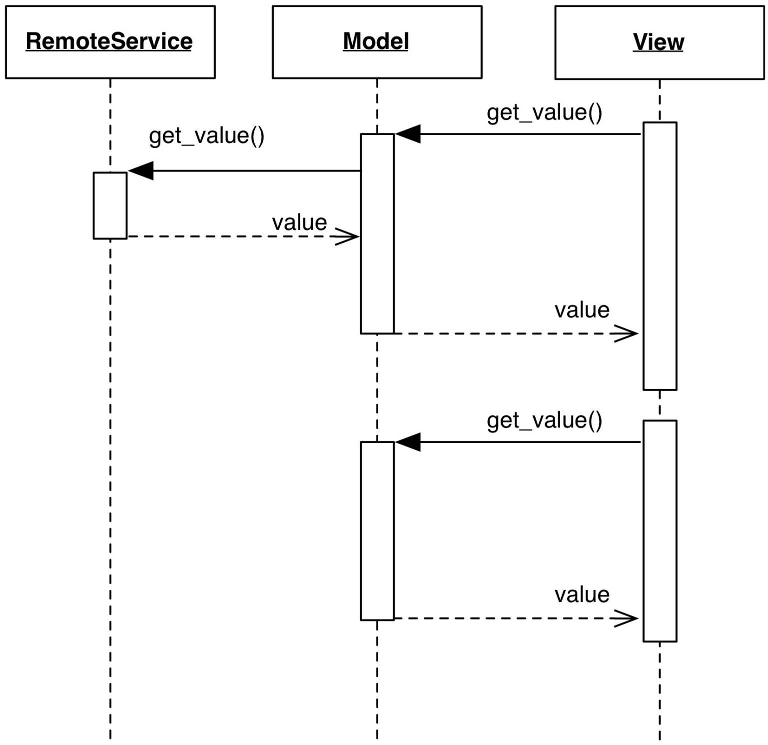
- Issue the request to the remote service
- Handle error conditions
- Convert received data into a representation useful for the client code.

Additionally, the Proxy Model can:

- Throttle an excessive number of requests in a short amount of time
- Cache obtained results and manage the cache expiration
- Observe the remote service for changes, for example through polling on a secondary thread.

Design

The design of a Proxy Model is generally dependent on the service it represents. Interfaces are designed to comply with the abstraction of the remote service. When the client code asks for data, the Model issues the appropriate request to the service provider. Client code is generally unaware of the exact nature of the backend exposed by the Proxy Model.



When the Model implements SQL access to a specific database table, the resulting Proxy Model is normally called **Table Data Gateway**. An instance of this Model represents the backend database Table as a whole, not a specific row, and its methods provide access to CRUD operations.

Practical example: Django Models

A simple Django Model provides an example of a Proxy Model.

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

For the class definition given above, Django's internals create a SQL table whose columns are defined by the class properties (`first_name` and `last_name`). Instances of the `Person` class are represented as table rows.

The Person class can be instantiated, stored in the database, modified, or used to retrieve an already present instance.

```
person = Person(first_name="A.", last_name="Einstein")
person.save()

person.first_name = "Albert"
person.save()

another_person = Person.objects.get(last_name="Galilei")
```

Under the hood, the Django Model implementation converts the operations into SQL statements.

Practical example: Thrift services

Another example of Proxy Model is provided by Thrift, a cross-language Remote Procedure Call framework. An Interface Definition Language specifies the interface of the remote service

```
service Calculator {
   i32 add(1:i32 num1, 2:i32 num2),
}
```

which is compiled into python to generate the following code

```
class Client(Iface):
  def __init__(self, iprot, oprot=None):
    self._iprot = self._oprot = iprot
    if oprot is not None:
      self._oprot = oprot
    self._seqid = 0

  def add(self, num1, num2):
    """
    Parameters:
     - num1
     - num2
    """
    self.send_add(num1, num2)
    return self.recv_add()

  def send_add(self, num1, num2):
    self._oprot.writeMessageBegin('add', TMessageType.CALL, self._seqid)
    args = add_args()
    args.num1 = num1
    args.num2 = num2
    args.write(self._oprot)
    self._oprot.writeMessageEnd()
    self._oprot.trans.flush()

  def recv_add(self):
    iprot = self._iprot
    (fname, mtype, rseqid) = iprot.readMessageBegin()
    if mtype == TMessageType.EXCEPTION:
      x = TApplicationException()
      x.read(iprot)
      iprot.readMessageEnd()
      raise x
    result = add_result()
    result.read(iprot)
    iprot.readMessageEnd()
    if result.success is not None:
      return result.success
    raise TApplicationException(TApplicationException.MISSING_RESULT,
                                "add failed: unknown result")
```

The above Proxy handles the complexity of the network exchange, providing a simple interface to client code

```
result = client.add(3, 4)
```

References

- Martin Fowler, "Patterns of Enterprise Application Architecture". Addison-Wesley, 2003.

- Django - https://www.djangoproject.com/
- Apache Thrift - https://thrift.apache.org/

Collection Model

Motivation

A Collection Model is an open-ended container of SubModels. It is similar in concept to the Compositing Model, but the similarities are only apparent. A Compositing Model:

- might be tailored to a specific number of SubModels.
- might hold SubModels of different types, handling and smoothing out their differences.
- aims at unifying data coming from its SubModels, so that their differences and separation are hidden to the View.

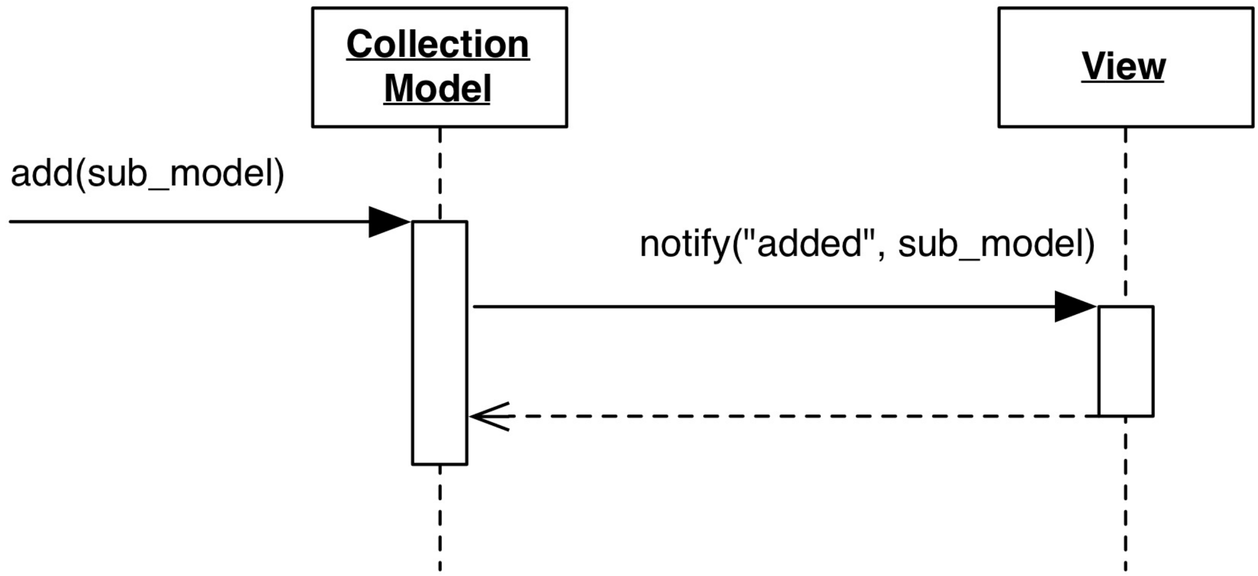
On the other hand, a Collection Model:

- Is an open-ended, potentially empty list of SubModels.
- The contained SubModels are of the same type.
- The Collection and its SubModels have independent nature and the objective of the Collection is not to hide the intrinsic separation of the SubModels, rather to refer to them as a group.

A trivial example of a Collection Model is an `Album` containing `Song` objects.

Design

The Collection Model is designed to notify its listeners when a SubModel is added, removed, or moved. In some cases, the collection model can also notify listeners when any of its SubModels changes.



More complex implementations may want to provide events and an interface for bulk addition, bulk removal, or bulk move of SubModels. Qualified messages can inform the listeners what kind of change has occurred, and which SubModels were affected.

Practical Example

Backbone.js Collection implements a Collection Model. The following code represents an example of the API described above

```
var album = new Backbone.Collection;

album.on("add",
         function(song) {
            alert("Song added callback executed.");
         }
);

album.add([
  {name: "Mozart collection"},
]);
```

References

- Backbone.js - http://backbonejs.org/

Caching Model

Motivation

Models may want to cache information when acting as a proxy for data sources that are slow, unreliable, or have an access quota or cost. Typical examples are network services, databases, disk access, or when the information is obtained from long-running computations.

Using a cache generally delivers the following advantages:

- Gives a faster response after the first access, improving the perceived performance of the application
- Neutralizes potential temporary failures of the data source, improving perceived reliability
- When accessing a remote resource, prevents repeated and frequent requests of the resource. Failure to implement caching in this case generally leads to a security response from the administrators of the remote resource (e.g. throttling, or denied access).

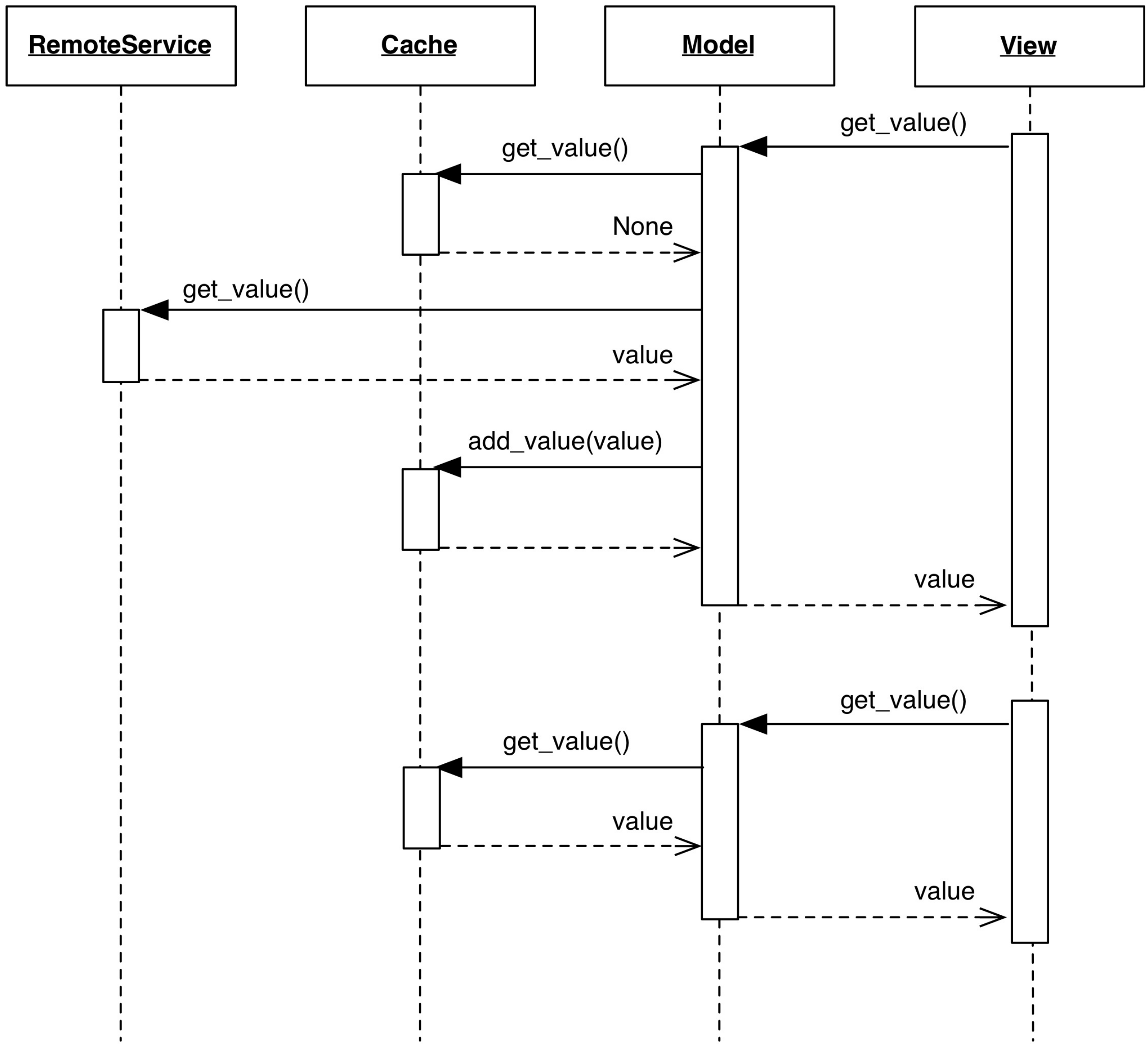
However, caching comes with a set of liabilities:

- Increases memory footprint of the application to store the cached data
- The Model may return outdated information to its client
- In multi-user web-applications, cached data may contain critical security information, and must therefore not be leaked to unintended recipients.

Design

The typical design of a Caching Model involves the following steps to retrieve the data from a source:

1. The View attempts to retrieve some information from the Model.
2. The Model first attempts to obtain this information from the Cache.
3. If the desired information is not found, the Model will perform a query to the data source, obtain the result, add it to the Cache, and return it to the View.
4. Additional attempts to retrieve the same information from the Model will extract data from the Cache.



When parameters are passed during the `get` request, the cache must have an index over these parameters to guarantee a retrieval of the intended information. For example, if a Model class `UserRepository` has a method `get_user_data(user_id)`, this method must use the parameter `user_id` to perform the intended retrieval from the cache.

To perform a `set` operation on the Model, two strategies are possible. The first one is to apply the changing action on the slow data source, followed by either a similar update or an invalidation of the cached data. This strategy has the following caveats:

- If the `set` action is expected to change the data source in a non-trivial way, it may not be possible to perform a sensible change to the cached data with the same business logic supported by the remote source. Instead, one should remove the cached data to promote a fresh retrieval at the next `get` operation.
- The operation requires a round trip to the slow data source, with consequences both at the UI level (failures must be reported, in-progress operations need to present a sensible progress bar, perceived performance may suffer) and also at the application design level (if repeated `set` operations are needed, performing a round trip to the

data source every time is wasteful and impacts performance).

The second possible strategy is to update the cached data and postpone the data source update, possibly bundling multiple changes into a single request. With this solution, the cache must not evict the changed information until it has been committed to the data source. Further issues may exist for resources that are shared among clients, or in case of application crash or network failure where the changed content is only partially submitted or not at all.

Clients of the Model may need to be aware of the existence of a cache if they want to guarantee up-to-date data, or need to force flushing of changes to the remote service.

Caching strategies

Choosing an appropriate caching strategy is a complex topic well outside the scope of this book. Several factors may need to be kept into account, such as data access patterns, specific requirements of the design and the application behavior, and data consistency.

To present a few among many possibilities, the most trivial strategy is to simply perform memoization of the result. A Model using memoization effectively uses a cache whose entries never expire. The cache size can be kept under control by discarding the Least Recently Used (LRU) data.

LRU constraint the size of the cache and keeps the most recently used data, but does not provide a predictable expiration strategy, something that may be required for some applications. For example, a News Ticker application retrieving information from a website may want to invalidate the cached data and perform a new retrieval, hopefully to obtain updated information, after a fixed absolute timeout (e.g. 15 minutes). Other applications may prefer a sliding expiration strategy: cached data expire only if not accessed within a given amount of time (e.g. 5 minutes, but the countdown resets to zero if requested again), or when an explicit method call is issued on the Model object. Smarter strategies may use a small payload request to ask the remote service if new or changed data are available, and act accordingly by either retrieving the new data or return the cached information.

Additional optimizations that are frequently used are preemptive caching and priority caching. Preemptive caching stores Model data that is likely to be accessed soon, even when not directly requested. For example, while retrieving information about a given user from a social networking website, a smart caching engine may decide to retrieve its most recent pictures and populate the cache of the picture Model objects; Priority caching allows to consider different levels of likelihood for eviction, removing low priority before high priority data.

Data/Metadata Model

Motivation

A Data/Metadata Model separates information in two categories:

- *data*: slow, heavyweight information that represents the content
- *metadata*: fast, lightweight information that describes the content

This distinction is often driven by the need for a Model object to present lean access to information needed for browsing, and resource consuming access to information needed for processing.

Design

The Model class implement getters for both data and metadata.

- Data methods retrieve the information lazily, possibly in chunks. Caching may not be possible if the size of the information is excessive.
- Metadata methods may or may not be lazily retrieved, and may use caching, especially when backed by network.

Clarity in documentation is essential to communicate the associated cost of retrieval.

Practical Example

An image processing program may have a `Movie` Model object to represent an image file on disk

```
class Movie(Model):
    def __init__(self, filename):
        self._filename = filename
        self._thumbnail = _extract_thumbnail(filename)
        self._length = _extract_length(filename)
        self._name = _extract_name(filename)

    # Metadata
    def thumbnail(self):
        return self._thumbnail

    # <similar getters for the remaining metadata>

    # Data
    def contents(self):
        # <...>
```

Properties such as `thumbnail`, `length` and `name` represent lean information useful for a movie browser. This information is extracted from the file at object instantiation and kept in memory.

The `contents()` method, on the other hand, retrieves the movie data directly from the disk and makes it available for additional processing (e.g. decoding and displaying).

Recording Model

Motivation

A Recording Model maintains a log of the changes occurred to its state. This information can then be used by external client code, for example to perform a refresh only if the visualized information was changed, or to perform undo.

This technique carries the liability of forgetting to clear the log, leading to memory exhaustion.

Practical Example

The following example shows a Model class that records changes to its properties

```
class Customer(Model):
    def __init__(self):
        self._changes = {
            "name": [],
            "surname": [],
            "address": []
        }
        self._name = None
        self._surname = None
        self._address = None

    def set_name(self, name):
        old_name = self._name
        self._name = name
        self._changes["name"].append((old_name, name))
        self.notify_listeners()

    # <similar code for set_surname/set_address

    def changes(self, property_name):
        return self._changes[property_name]

    def clear_changes():
        for changes in self._changes.values():
            del changes[:]
```

The setters record the changes in the `self._changes` dictionary. Performing the following operation

```
c = Customer()
c.set_name("Rob")
c.set_name("Robert")
```

will produce a `self._changes["name"]` list containing two elements: the first transition `(None, "Rob")`, and the second transition `("Rob", "Robert")`.

Variation: record the sequence of changes

The solution given above does not record if, for example, the surname was changed before or after the name. An alternative implementation can store this information by e.g. using a list instead of a dictionary for `self._changes`.

```
class Customer(Model):
    def __init__(self):
        self._changes = []
        # <...>

    def set_name(self, name):
        # <...>
        self._changes.append(("name", old_name, name))
        # <...>

    def changes(self):
        return self._changes
```

Variation: interest only in the last change

The log length can be limited, potentially to a single change, by simply replacing the previous change information with the latest. This approach also removes the liability of forgetting to clear the log.

```
class Customer(Model):
    de f __init__(self):
        self._last_change = None
        # <...>

    def set_name(self, name):
        # <...>
        self._last_change = ("name", old_name, name)
        # <...>

    def last_change(self):
        return self._last_change
```

Transactional Setting

Motivation

Your Model has multiple settable attributes, each one listened independently by Views. The business logic requires setting two or more attributes to reach the desired final Model state. Each set operation triggers a notification, as shown:

1. Set A attribute on the Model
2. Views are notified of change
3. Set B attribute on the Model
4. Views are notified of change

During step 2, listeners will be notified of the change and sync against a Model where only one of the attributes has been changed. Depending on the specific details of your Model and Views, this state may be inconsistent or not representable by the Views. However, setting and notification of the individual attributes may still be needed for specific use cases.

The desired goal is to trigger notification only when all set operations have been performed, as follows:

1. Set A attribute without notification
2. Set B attribute without notification
3. Views are notified of change

Design

Transactional setting is the most trivial strategy to achieve the result outlined in the Motivation, the others being using a Lazy Model or an Accumulator, examined later.

Transactional setting implements a setter function accepting both attributes. This setter function alters the object state and then issues the change notification.

Practical Example

A Model representing a network connection to a remote host might want to present functionality to change the host, the port, and both simultaneously. If any listener is in charge of initiating a network connection, the notification should be delivered only when the Model contains correct host information.

```
class Connection(Model):
    def set_host(self, host):
        self._host = host
        self.notify_listeners()

    def set_port(self, port):
        self._port = port
        self.notify_listeners()

    def set_host_and_port(self, host, port):
        self._host = host
        self._port = port
        self.notify_listeners()
```

Without the `set_host_and_port()` method, reconnecting from `http://one.example.com:80` to `http://two.example.com:8080` would either trigger a rogue connection to `http://one.example.com:8080` (when `set_port(8080)` is called) or to `http://two.example.com:80/` (when `set_host("two.example.com")` is called).

For obvious reasons, the `set_host_and_port()` cannot reuse the independent setters, and must therefore reimplement the setting logic. If this logic is complex, it might be good practice to factor it out in a separate "silent setter" method.

Variation 1: qualified notification

If the Model performs qualified notification, the behavior in absence of transactional setting will be

1. Set A attribute on the Model
2. Views are notified of A change
3. Set B attribute on the Model
4. Views are notified of B change

Transactional setting will provide a method performing the following sequence:

1. Set A attribute on the Model
2. Set B attribute on the Model
3. Views are notified of A change
4. Views are notified of B change

or if the qualification protocol allows to deliver information about multiple changes at once

1. Set A attribute on the Model
2. Set B attribute on the Model
3. Views are notified of A+B change

Variation 2: set/add multiple items

Transactional setting can also be used to add multiple elements at once, generally for a Model behaving as a list. The Model might want to implement an interface for adding both one or multiple elements. The implementation normally acts as a syntactic sugar to generate a list with one item.

```
class BookCollection(Model):
    def add_book(self, book):
        self.add_books([book])

    def add_books(books):
        self._books.extend(books)
        self.notify_listeners()
```

Variations on the notification strategy

- Qualified Notification: Deliver notification with enhanced semantics to inform listeners about what has changed.
- Qualified Notification Model with Subscribing: View subscribes to specific events and gets notified only when they occur.
- Passive Model: A Model without notification features.
- Lazy Model: A Model delivering its notifications on explicit request.
- Accumulator: Listens to submodels and squashes multiple notifications into a single one.
- Pre/Post notification: Deliver notifications before or after the change.
- Vetoers: Inquire listeners to approve or deny a change to occur.
- Signals: Isolate notification into a separate object.

Qualified Notification

Motivation

The most basic form of Model notification is unqualified: Views are just informed that a change occurred in the Model state. In response, Views retrieve the new Model state and repaint themselves accordingly. This approach is simple but coarse. Consider these scenarios:

- A specific View is only interested in a subset of the Model. The Model changes, but not in the data relevant to the View. The View is forced to repaint even if none of the data it displays actually changed
- A View has additional state which is destroyed by a full repaint. For example, a TreeView representing files in a directory keeps state for the opened/closed sub-branches. If a new file is added and the View is forced to rescan and repaint, the open/closed state is discarded. Knowing the nature of the change would help in handling this case.
- A View takes a very long time to perform a repaint from the full Model's content, but it can run faster if it can operate by knowing only the change.

A Qualified Notification addresses the above cases by enhancing the notification system with a more fine-grained protocol carrying information about what has changed in the Model.

Design

Qualified Notification can be implemented by passing arguments to the Views' `notify()` method. When the Model changes and calls this method on the subscribed Views, it passes information about

- The subject of the change (*e.g.*, which Model property has changed)
- The nature of the change (*e.g.*, the previous and new values)

The following example implementation trivially satisfies the above design

```
class View():
    def notify(self, name, old_value, new_value):
        # ...
```

Unfortunately, this solution does not allow to notify about more than one property change at a time. Additionally, it does not support multiple Models notifying the same View, as the View would not be able to differentiate which Model is reporting the change.

A more flexible approach is given by the following implementation

```
class View():
    def notify(self, change_info):
        # ...
```

where `change_info` is a dictionary or other data object describing the change in enhanced detail. Models are responsible for creating and populating this data object so that is meaningful to the receiving View. As a side effect of the design, the View does not need to inquire the Model's state anymore. Instead, it keeps its synchronization by means of the change information.

Most modern MVC frameworks provide qualified notifications in one form or another (e.g. bound properties or signals). An unqualified reporting is uncommon but not completely dismissable as a strategy, especially for cases where there is no granularity in the Model to be exploited, or the View cannot take advantage of a fine-level notification.

Qualified Notification with Subscribing

Motivation

In this variation of Qualified Notification, the listener specifies which change it is interested in, and a notification is issued only with the relevant change occurs. The two solutions therefore differ in which entity performs filtering of unnecessary notifications: in Qualified Notification the filtering is performed by the listener, after the notification is delivered, while in Qualified Notification with Subscribing, the filtering is performed by the Model, honoring the listener's registration instructions.

Design

For a Qualified Notification with Subscribing, the Model must support:

- an appropriate signature for the `register()` method, allowing to specify the type of modification a listener is interested in.
- preservation of the passed information in an internal registry, which will be used during notification.

When the Model experiences a change in its content, it must consult its internal registry to verify if a listener is interested in this information. The Model then informs the listener only if this is the case.

Practical Example: Apple KVO (Key-Value Observer)

Apple KVO (Key-Value Observer) is a form of Qualified Notification with Subscribing. The Model supports an `addObserver` method, whose signature allows to specify the property the listener is interested in, and the type of change. For example, a listener registers for changes on the Model property `my_property` with the following call

```
[model addObserver:destination
       forKeyPath:@"my_property"
       options:NSKeyValueChangeNewKey
       context:nil];
```

When the `my_property` value changes, the listener's method `observeValueForKeyPath:ofObject:change:context:` will be invoked on the listener. Changes in other Model's properties will not be delivered.

References

- Key-Value Observing Programming Guide
- KVO considered harmful

Passive Model

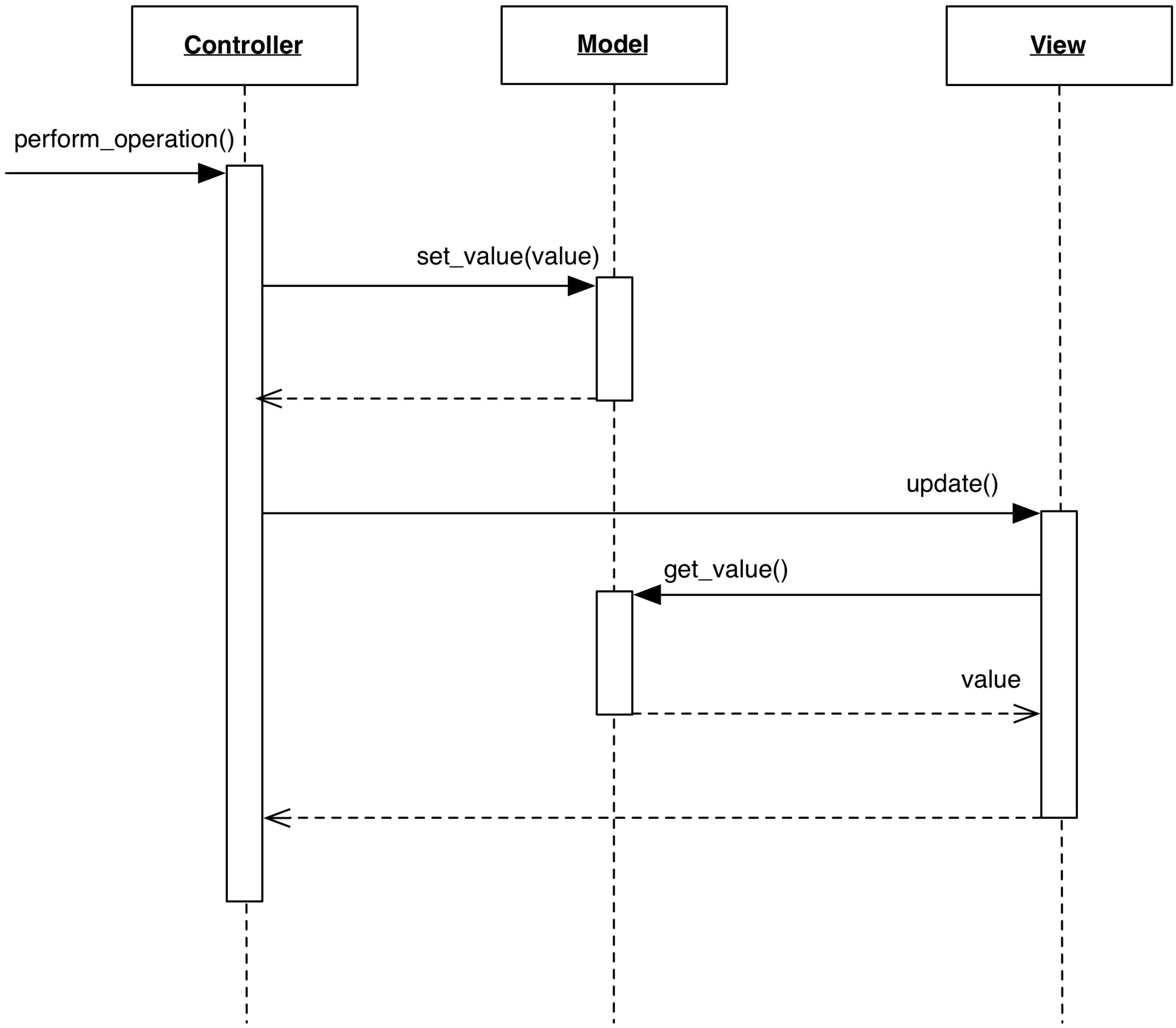
Motivation

Traditional MVC uses the so-called **Active Model**: when the Model changes its listeners are notified of the change. Its counterpart is the **Passive Model** which does not perform notification services. Instead, this task is performed by the Controller.

Passive Model has its area of excellence in Web-based MVC, where the fundamental nature of the HTTP protocol prevents the Model to notify the View. It is also relevant for all those cases where notifications cannot be produced because the data source is unable to provide this service, such as most databases or a plain file.

Design

The interaction diagram shows the behavior of a Passive Model



Specifically, the sequence of events is:

1. The Controller modifies the Model
2. The Controller informs the View to update itself
3. The View now inquires the Model contents as in the Active case.

The major shortcoming of a Passive Model is that Views get desynchronized if multiple Controllers can modify the Model. Collaboration between Controllers can prevent this desynchronization, but for all practical purposes an Active Model quickly becomes a better solution. If this is required, a Passive Model can become Active either through inheritance or by using a wrapper class satisfying the Passive Model's original interface. This wrapper will receive change requests from Controllers, delegate them to the Passive Model, and finally notify the listeners.

Despite the disadvantage, the Passive Model has the following important advantages:

- Any object (e.g. a key-value dictionary, a list, a single value, a previously developed business object) can be used as a Model without modifications
- It allows better control on the refresh of the View. The Controller can issue multiple

changes to the Model without triggering a View's refresh for each change
- As commented in the Motivation section, Web-based MVC cannot use an Active Model strategy due to the intrinsic separation of Models from rendered Views.

Practical Example: Django MVC

The web framework Django is our choice to present a simple case of Passive Model. Please note that this example is not meant to be considered orthodox Django style, and has been stripped beyond the bare minimum to illustrate the topic.

A Model in Django is specified as a python class with appropriate descriptors

```
class Article(models.Model):
    title = models.CharField(max_length=200)
    text = models.CharField()
```

This model is backed by a database, which stores the actual state. It is unable to deliver notification in case of database content changes, and is therefore a Passive Model. In the trivial case of Web MVC here presented, the need for notification does not emerge because the refreshing of the View is triggered by an explicit browser request. The Controller handles this request, retrieves the Model, performs any required action, and renders the View with the updated data.

```
def modify_article(request, article_id):
    article = get_object_or_404(Article, pk=article_id)

    # These changes do not trigger any notification. The model is passive.
    article.title = request.POST["title"]
    article.text = request.POST["text"]

    # Persist the replaced data.
    article.save()

    template = Template("<html><body>"
                        "<h1>{{ article.title }}</h1>"
                        "<p>{{ article.text }}</p>"
                        "</body></html>")
    context = { "article" : article }
    html_doc = template.render(context)

    return HttpResponse(html_doc)
```

References

- MSDN documentation: Model-View-Controller. Passive Model

Lazy Model

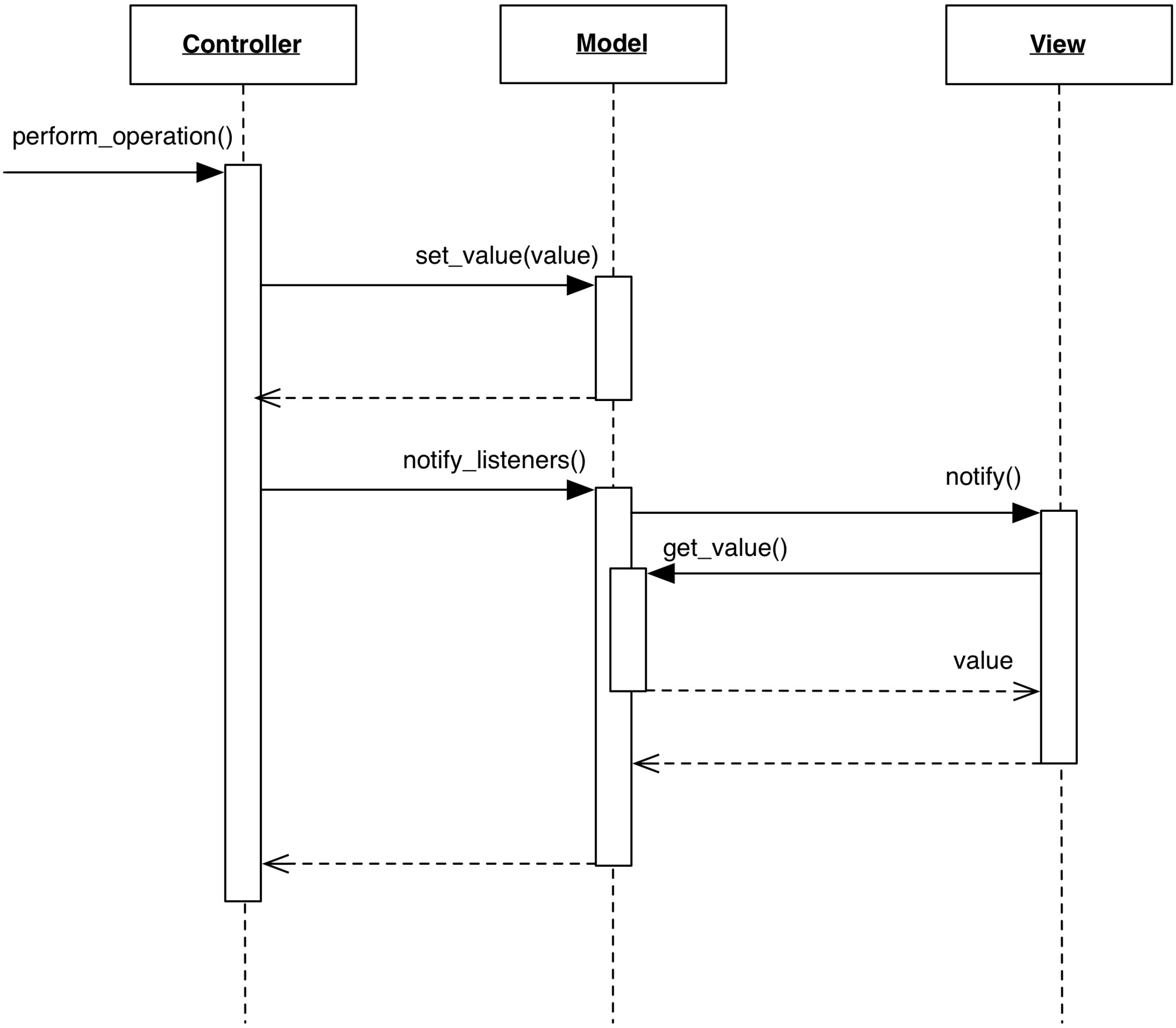
Motivation

A Lazy Model is an intermediate solution between Active and Passive Model that retains centrality of the notification bookkeeping, but delegates notification triggering. It is a good strategy to retain full control on the notification flow as in the Passive model, but at the same time keep the listeners' list centralized so that they can all be notified of the changes, no matter which Controller performs them.

With the resulting design, the Controller is free to perform multiple changes on the Model object, and trigger the notification when done.

Design

Like an Active solution, listeners register on the Model and await for notification; Differently from it, methods that modify the Model do not call `Model.notify_listeners()`. Instead, the Controller issues the call.



Practical example

A trivial example of Controller code using a Lazy Model notification strategy would behave as follows

```
class Controller():
    def perform_operation(self):
        model = self.model

        # None of these setters trigger notification.
        model.set_name("Enrico")
        model.set_surname("Fermi")
        model.set_nationality("Italian")

        # Views are notified of the changes here.
        model.notify_listeners()
```

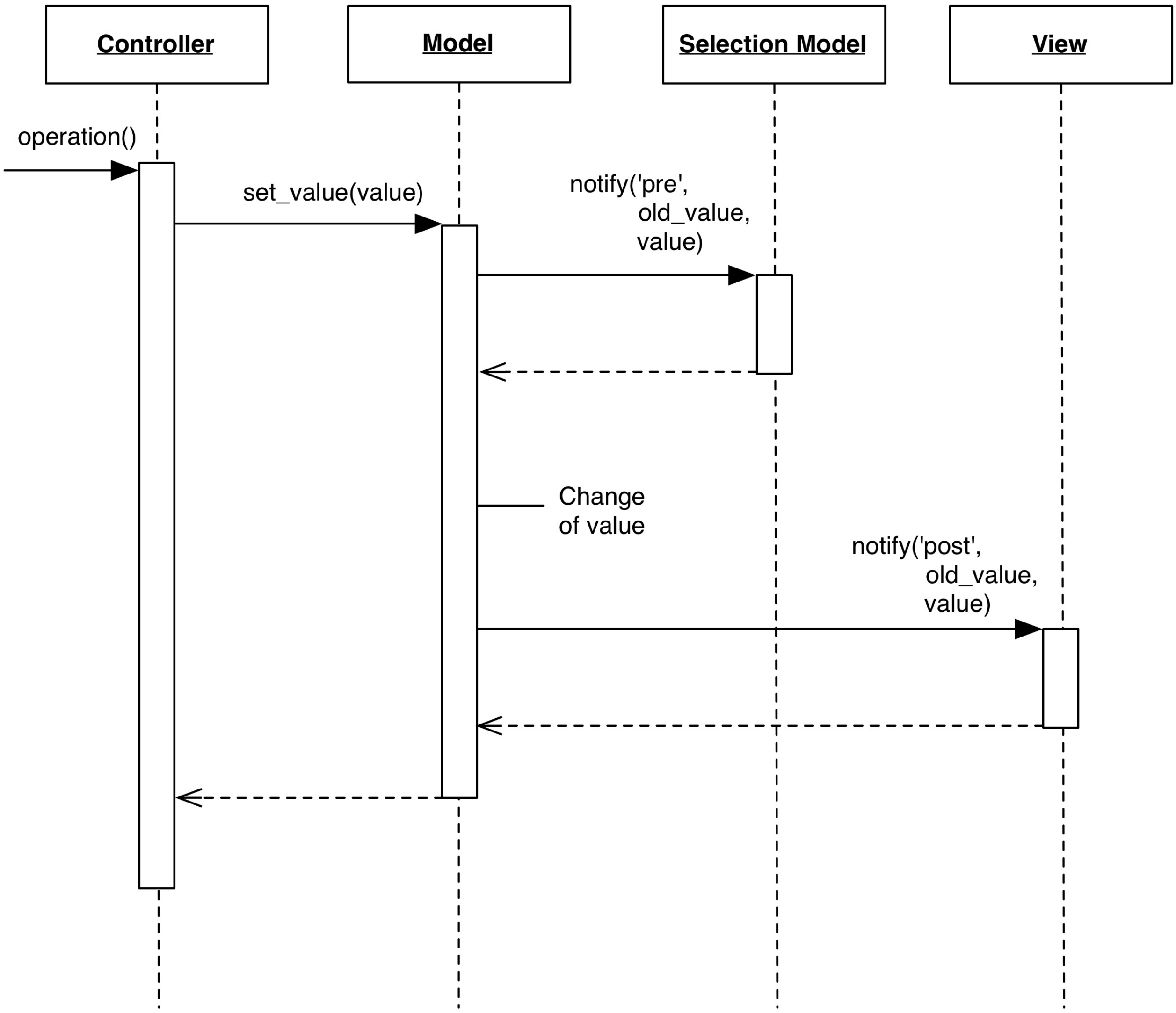
Pre/Post notification

Motivation

The Model wants to inform its listeners not only when data have been changed, but also just before changing them. A common reason behind this need is to support vetoing, but other motivations may exist.

Design

To implement Pre/Post notification, the Model simply issues two different notifications, one before changing the Model state, and one after.



These notifications must be qualified, otherwise the interested listeners would not be able to differentiate the pre-vs-post nature of the received notification. For example, a Model could issue `about_to_change`, apply the change, and finally issue `changed`. Both the old and new

value are also generally passed as part of the notification.

Practical Example

Although other approaches could be used to handle this specific use case, pre-notifications can be useful to keep a Selection Model synchronized when the View in not collaborative in handling a selection referring to a non-existent entry.

Our example has a Model containing a list of items, and a Selection Model holding information about which items in the Model are selected. If the View cannot handle selection of items not present in the Model, the following scenario would break the View:

1. An item of the Model is currently selected. This implies that an entry for this item is present in the Selection Model.
2. A Controller issues a request to the Model to remove the item. Note that the controller may not be aware of the existence of a Selection Model, so it is unable to deselect the item to prevent this scenario.
3. The Model removes the item, and notifies the View of its change.
4. The View renders the Model, followed by rendering the Selection, but the selected item is no longer in the Model. The View cannot handle this situation and issues an exception.

Using a pre-notification can work around the View implementation detail:

1. A Controller issues a request to the Model to remove the item.
2. The Model issues an "about to remove item" pre-notification.
3. The Selection Model listens and receives the Model pre-notification.
4. The Selection Model reacts by removing the item from its internal state.
5. The Selection Model change triggers a notification to the View which is handled by visually deselecting the item. The Model hasn't changed yet, so the View is retrieving valid data.
6. Finally, the Model removes the item, and post-notifies the View to re-render itself with one less item.

Through pre-notification, the View retrives consistent data throughout the operation. Without the existence of the pre-notification, we would have to rely on the order of notification of the post-notification to notify the Selection Model first, then the View. In general, it is not good practice to rely on ordering of different listeners on the same notification channel, because the order may not be guaranteed by the notification framework.

Signals

Motivation

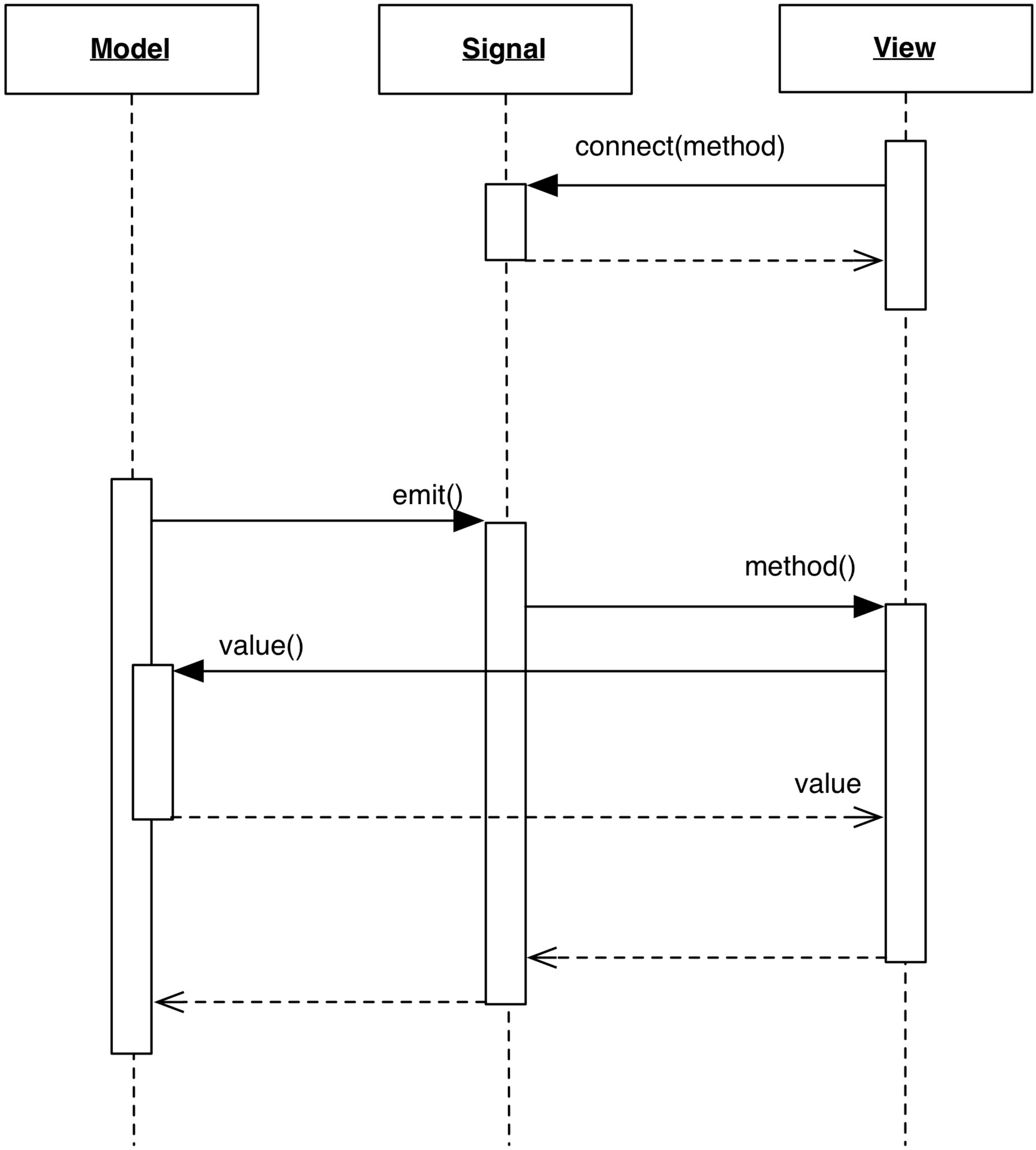
A problem carried over from the traditional MVC approach is the need for the Model to hold a collection of its listeners for notification purposes. This introduces a loose dependency between the Model and the listener interface that all Views must implement.

The design presented in this section decouples this dependency between Models and Views, extracting the notification system in a separate object, the Signal, acting as a relay between the two. Althought not immediately apparent, the benefits of this approach are important:

- Different Signals can represent different changes, providing the flexibility of a Qualified Notification.
- Signals can be made to call specific View's methods, instead of a generic `notify()`, passing specific arguments.

Design

A Signal class holds the notification infrastructure previously held by the Model: a list of listeners and methods for listeners to register and unregister. Notification to listeners is triggered via a `Signal.emit()` method.



The Model defines its signalling capabilities by exposing member variables of type Signal. These member variables are given appropriate names to convey the nature of each reported event. Listeners subscribe to the signals they are interested in.

When a change of the appropriate type occurs, the Model triggers the notification by calling `emit()` on the Signal object, which in turn will notify each individual listener.

Variation: additional flexibility to notification

A basic implementation just notifies the listeners. Increased flexibility can be obtained by allowing the passing of arguments to the `emit()` method, allowing further qualification of the emitted signal.

The Signal can be made configurable in its functionality. A Signal class could for example implement three strategies for notification:

- open: the notification is performed as soon as triggered.
- closed: the notification is silenced.
- hold: the notification is not delivered, but it is retained for later (i.e. an Accumulator-like behavior)

Practical examples

A Signal implementation conforming to the above design can be found in Boost.Signal2. In the following example, we demonstrate the registration of an arbitrary method `print_sum` to a Signal object and its execution when the signal is emitted. The `emit` functionality is implemented through `operator()`. Note how parameters can be passed to the connected functions.

```
void print_sum(float x, float y) {
    std::cout << "The sum is " << x + y << std::endl;
}


void main() {
    boost::signals2::signal<void (float, float)> sig;

    sig.connect(&print_sum);

    // emit
    sig(5., 3.);
}
```

Another example of Signal design is Qt Signal/Slot mechanism, although the internals do not make use of a literal Signal object, and the mechanism requires the support from a special preprocessor

```
class Model : public QObject
{
    Q_OBJECT

public:
    Model() { value = 0; }
    int value() const { return value; }

public slots:
    void setValue(int new_value) {
        if (new_value != value) {
            value = new_value;
            emit valueChanged(new_value);
        }
    }

signals:
    void valueChanged(int new_value);

private:
    int value;
};
```

The special signal `valueChanged()` is emitted every time a new value is set. Receiving objects can subscribe to the Signal and guarantee synchronization

```
Model *model1 = new Model()
Model *model2 = new Model()
QObject::connect(model1, SIGNAL(valueChanged(int)), model2, SLOT(setValue(int)))
model1->setValue(42)
```

In the above code, `model2->setValue` will be called when model1 `valueChanged` signal is emitted. This will synchronize the two objects to hold the same value.

Practical example variation: notifying Views

One important feature of Qt signals is that they are not only found on Models, but also on Widgets and Views. Buttons, for example, emit a `clicked()` Signal when clicked. Signals can therefore be used as a generic notification mechanism applicable to any context where decoupling is desired.

Variations on the View

- Pluggable View: Client code injects data extraction logic in the View's at initialization.
- Passive View: View's state is fully orchestrated by the Controller.
- Widget level vs. Container level View
- Visual Proxy: The Model acts as a factory of its own View, choosing the appropriate

```
representation for its data.
```

- Data Dialog: "Fire and forget" dialog to accept information from the user.
- Visibility allowed notifications
- Synchronization View state

Pluggable View

Note: In the context of the Web framework Flask, Pluggable View refers to a different concept.

Motivation

For some applications, different Views may have the same implementation for the UI part, but different implementation for data extraction from the Model. To prevent duplication, two strategies has already been analysed:

1. implement the common UI logic in a base View class, and have subclasses reimplementing the data extraction logic.
2. use a Value Model, which puts the extraction logic on a Model adapter object, transforming the Model's complex interface into a trivial getter/setter pair. The resulting View is agnostic of the nature of the Model and the complexity of the extraction logic.

In this section, we present the **Pluggable View** design. An adaptor function is defined, and injected into the View at initialization. The model is processed through this function to obtain the data to be displayed.

Design

The View accepts a function object containing the data extraction logic. This function receives the Model object and returns the relevant data needed by the View.

```
def extractor(model):
    return str(model.get_value())

view = View(model, extractor)
```

The View listens to the Model for change notifications. When a change occurs, the View calls the function object, passing the Model and obtaining the relevant information as a return value.

```
class View:
    # <...>
    def notify(self):
        value = self.extractor(self.model)
        self.set_value(value)
```

Variations of this solution allow for multiple functions to extract different parts of the Model, or the possibility to swap the function object after construction. More elaborate extractors can return "UI decorated" information, such as returning both the value and the text color to use.

Passive View

Motivation

A major issue arising from a complex View is automated testing. Visual components require User interaction, which depends on the creation of events programmatically and the execution of an event loop to dispatch them. Both requirements may present challenges.

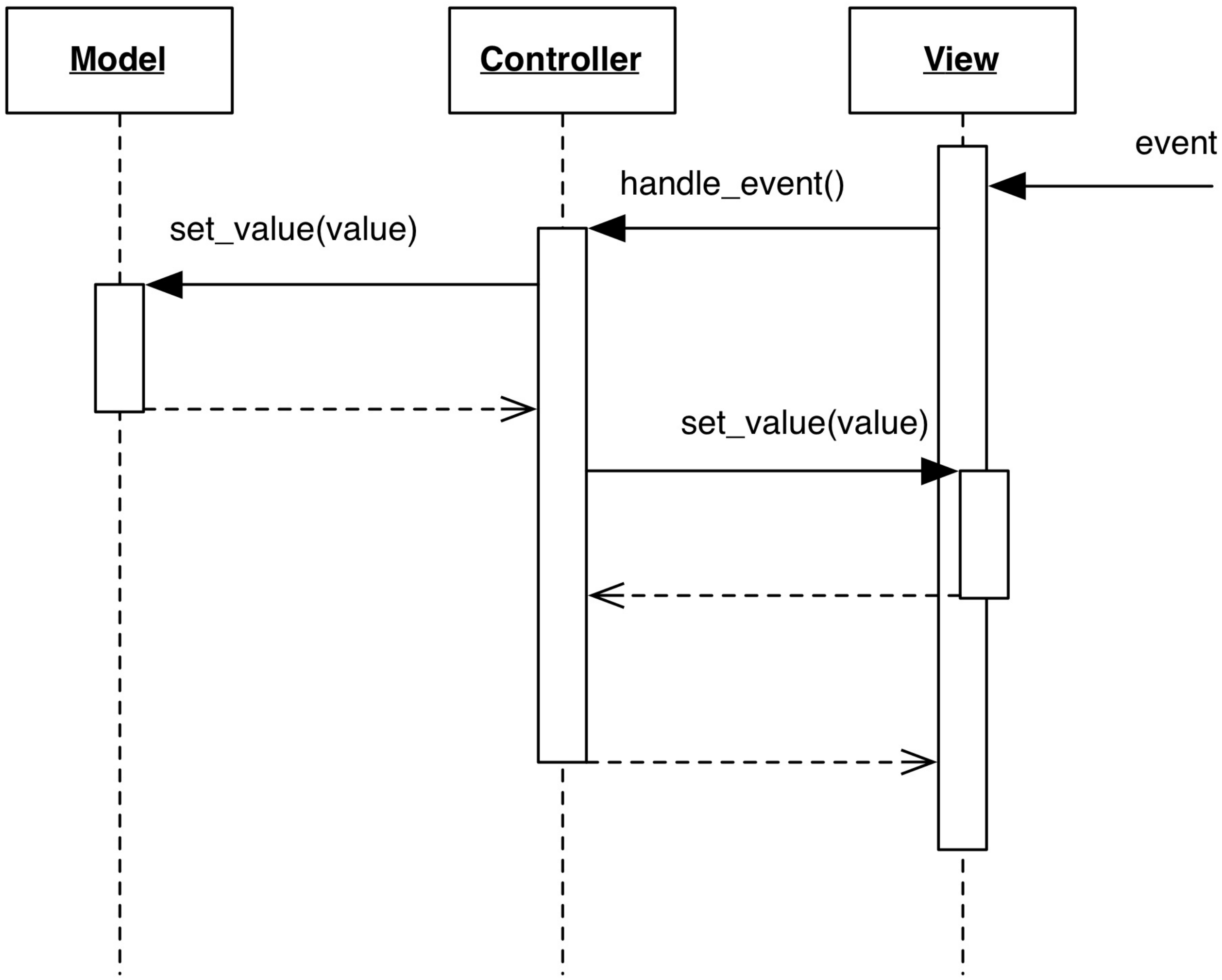
The **Passive View** (also referred as **Humble Dialog** or **Humble View**) eases testability by moving all View logic from the View to the Controller. This results in a View humble in logic and with no awareness of the Model. In fact, the View is not subscribed to Model changes, and is not extracting data from it. It is instead the Controller's duty to push data into the View, which is assembled from off-the-shelf widgets.

Design

In a Passive View design, all logic goes in either the Controller or the Model. In particular, the Controller is in charge of

- Visual logic (e.g. altering colors of widgets)
- Visual semantics (interpreting the meaning of a Button click)
- Data validation (checking if data from the View are correct, potentially with support from the Model)
- Synchronization of the View's contents, through the View's accessor methods.

The following interaction diagram shows the central role of the Controller in the synchronization, and the lack of interaction between the View and the Model.



which can be explained as follows:

1. When the View receives user events, they are forwarded to the Controller as in Traditional MVC
2. The Controller acts on the Model
3. Either immediately (if Passive Model, in the Figure), or in response to a Model notification (if Active Model), the Controller updates the data displayed by the View's widgets, synchronizing them against the new Model contents.

The View is empty of logic, and can be completely replaced with a mock object implementing the same interface. When testing the Controller, the mock View's methods will be called, and their invocation can be checked by the test. The mock View can also be set up to present specific data to the Controller, simulating User input.

Widget-level vs Container-level MVC

Motivation

In our previous exploration we defined Views without any detail on the scope of their implementation. Given a window containing a dozen of widgets, two implementations of the View object are possible:

- The View is the full window, forming a single MVC Triad through a complex, dedicated Controller, or
- Each widget is considered a View. There are multiple independent triads, one per each widget.

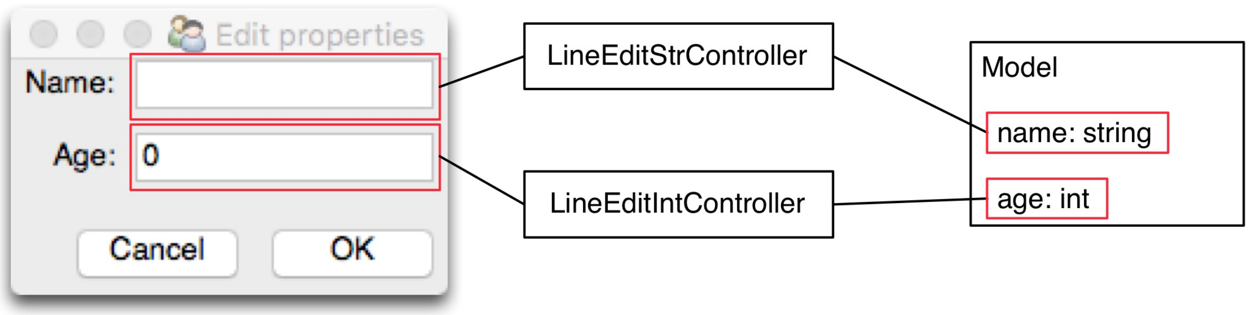
Both approaches are possible, and they are called **Container-level** and **Widget-level MVC**, respectively.

Each solution has advantages and disadvantages, and the consequence is that a mixture of both approaches is used: Widget-level has the strong advantage of reusability. Views are off-the-shelf UI widgets, and Controllers are generic classes tailored to specific widgets. The connection between the widget and the Model property become straightforward, albeit prone to produce boilerplate code to set up the connection. With some additional support, the boilerplate can be eliminated, leading to the design known as Data Binding. For this reason, widget-level MVC is the most frequently used building block because of its strong off-the-shelf nature.

On the other hand, a Container level approach is useful when there's need for a specialized Controller handling multiple Models and their data's cross validation, and for a View that acts as a container of complex SubViews.

Design

A Widget-level design requires one specific controller for each Widget of the toolkit. The role of each Controller is to handle the communication and data conversion between the widget and a specific property of the Model, and vice-versa. In particular, the Controller must push changes onto the Widgets, which behave as Passive Views.



Considering that Model properties can have different types, a strategy for correct type handling may be required as well. This gives raise to a potentially large set of controllers for all the possible combinations widget/model type (e.g. `ListEditStrController` , `ListEditIntController` , and so on). An alternative approach is to have only per-widget controllers (`ListEditController`) which accept a type conversion strategy as a conversion function. Introspection is then performed on the Model property to determine its type, followed by a lookup of the appropriate conversion strategy from the View representation (e.g. a string) to the actual Model type (e.g. an int).

Synchronization between Model and Widget is generally bi-directional: if the Model changes, the Widget will update (Model to Widget), and if the Widget is updated by the User, the Model will change (Widget to Model). There are however cases where one-directional communication is preferrable: "Model to Widget" only is the reasonable choice when the Widget is read-only. "Widget to Model" only is instead used when coercion of the type introduces bounceback from the Model. This particular problem will be explored in more detail later (see Coercion Bounceback).

Practical example

is better explained with a practical example supporting the explanation: a window contains a LineEdit widget to write a message and a "Send" button to send the message. The resulting design will have two widgets (LineEdit and Button, each of them a View), two specialized Controllers (LineEditController and ButtonController) and a single Model.

: a generic LineEdit could be connected to a Model string variable via a LineEditStringController, or to a float variable via a LineEditFloatController. The Controller would take care of validating and converting the data type (for example, from the string provided by the LineEdit to a float)

but it introduces the following shortcomings:

- The controller may be too trivial in some cases, in particular with complex Models (e.g. multiple instances must be handled) or complex Views (e.g. different widgets that need to be analyzed by the controller at the same time).

Visual Proxy

Motivation

Visual Proxy is an approach first presented by Allen Holub in 1999. It it is wildly different from the previous approaches, but definitely contains interesting ideas. Holub argues the following:

- It is very rare for the same Model to be represented at the same time in two different ways
- Model representation is not about the Model object per-se, but for some specific attributes of that object. These attributes can be presented with a well defined UI widget, regardless of where they will appear in the View. For example, `Person.name` is invariably represented as a string, and its View presentation is an editable textfield widget throughout the application
- All Model designs generally assume a get/set approach to Model state modification. From a strict Object Oriented interpretation, getter/setters is a *faux pas* and should be avoided, as they are just access to private state in disguise
- Essential separation between Model and View is clumsy to achieve in MVC, which does not scale well at the application level
- An Object Oriented system should focus on objects issuing behavior requests to one another, with the data staying put. The previous designs instead focus on data transfer from the view to the model, and *vice-versa*.

Out of these considerations, the proposed alternative is to let Models create their own Views, according to the data they hold. For example, a Model containing a string, a date and an integer range return a View containing a LineEdit, a Calendar and a Spinbox constrained to the the appropriate range.

This approach, while appealing in its cleverness, is not without shortcomings. Responsibilities that are traditionally handled by the View layer are now handled, directly or indirectly, by the Model:

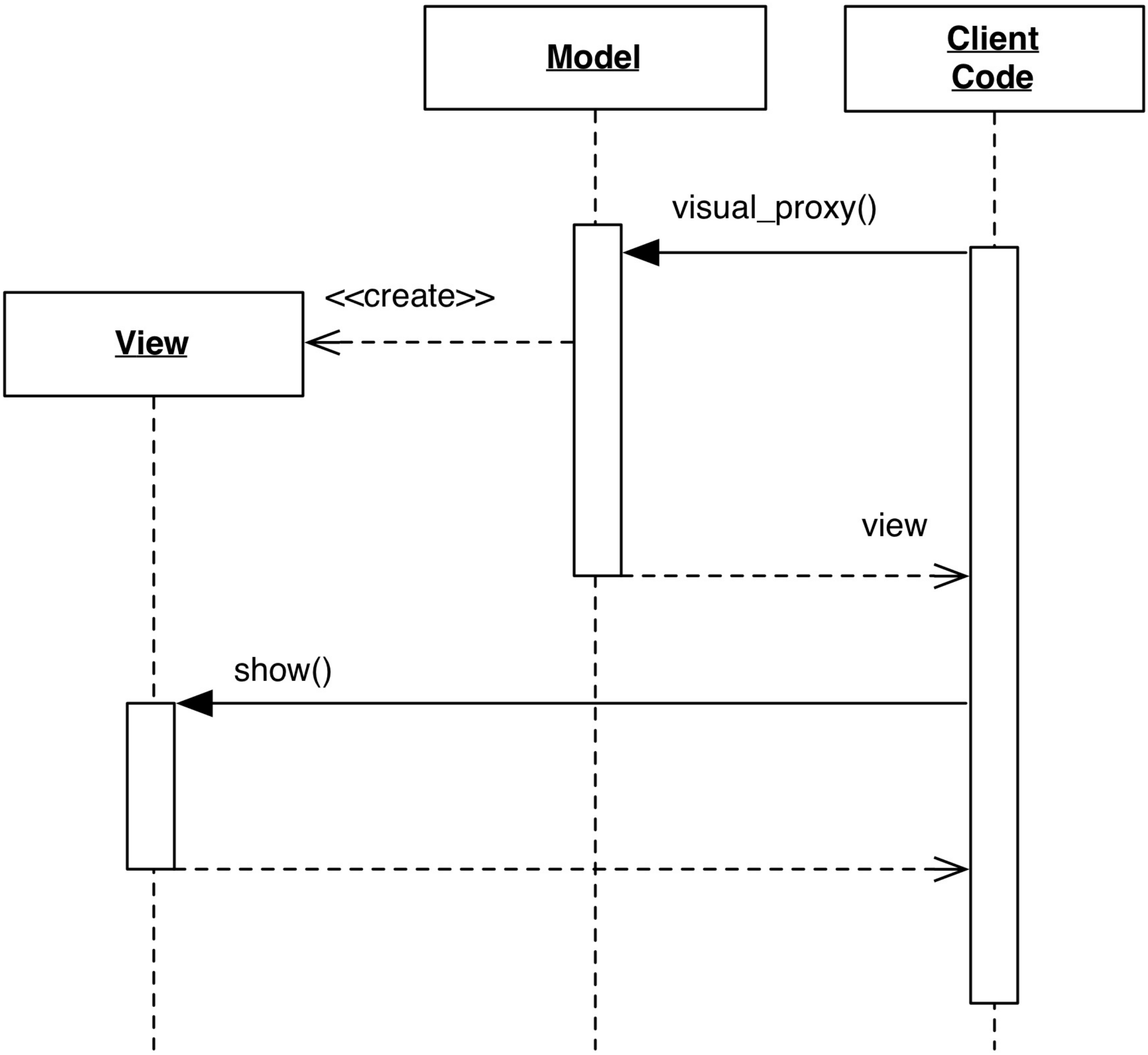
- The Model layer has a dependency on the GUI toolkit. This may have deep implications for testability, reuse, and requirements for the Model layer
- If the Visual Proxy contains static parts (such as a "Name" label followed by the line editor to input the name) the Model objects have to handle localization of the "Name" string in other languages
- Logical dependencies between visual components (*e.g.* when this value is 1, enable that checkbox) must also be moved to the Model layer.

Note that this approach is different from a UI Retrieving Model. The latter considers the User as a source of data, and generates a short-lived UI for that specific retrieval. Visual Proxy, on the other hand, is a full fledged foundation of all Model objects to provide their own View as implicitly described by their data.

Visual Proxy should also not be confused with View-aware Model, as they handle different concerns: Visual Proxy deals with View's creation. View-aware Model deals with Model-View synchronization.

Design

Model objects act as factories for the UI of their own attributes



The resulting message flow is simplified: all the data synchronization happens between the Visual Proxy and its backend attribute. The Client code is not concerned with this transaction. It just coordinates creation and presentation of the Visual Proxy object to the User.

Practical Example

The following practical example represents a boilerplate implementation of the design, for clarity purposes. Two methods are provided to generate a Visual Proxy: `visual_proxy_attribute` returns the Proxy for a specific attribute of the Person Model class; `visual_proxy` returns instead the full UI representation of the Model properties.

For simplicity of presentation, this example only performs synchronization in one direction, from the View to the Model. Changes in the Model are not updating the View.

```
class Person(object):
    def __init__(self, name="", surname="", age=0):
        self._name = name
        self._surname = surname
        self._age = age

    def visual_proxy(self, read_only=False):
        view = QWidget()
        layout = QGridLayout()
        for row, attr in enumerate(["name", "surname", "age"]):
            layout.addWidget(QLabel(attr), row, 0)
            layout.addWidget(self.visual_proxy_attribute(attr, read_only), row, 1)
        view.setLayout(layout)
        return view

    def visual_proxy_attribute(self, attribute, read_only=False):
        method = getattr(self, "_create_{}_view".format(attribute))
        return method(read_only)

    def _update_name(self, name):
        self._name = name

    def _update_surname(self, surname):
        self._surname = surname

    def _update_age(self, age):
        self._age = age

    def _create_name_view(self, read_only):
        if read_only:
            widget = QLabel(self._name)
        else:
            widget = QLineEdit()
            widget.setText(self._name)
            widget.textChanged.connect(self._update_name)

        return widget

    def _create_surname_view(self, read_only):
        if read_only:
```

```
            widget = QLabel(self._surname)
        else:
            widget = QLineEdit()
            widget.setText(self._surname)
            widget.textChanged.connect(self._update_surname)

        return widget

    def _create_age_view(self, read_only):
        if read_only:
            widget = QLabel(str(self._age))
        else:
            widget = QSpinBox()
            widget.setValue(self._age)
            widget.valueChanged.connect(self._update_age)

        return widget

    def __str__(self):
        return "Name: {}\nSurname: {}\nAge: {}".format(self._name, self._surname, self
._age)
```

Client code can now request the Visual Proxy by issuing:

```
person = Person(name="John", surname="Smith", age=18)
view = person.visual_proxy()
view.show()
```

Practical Example: Enthought traits

A more streamlined approach to the Visual Proxy design is provided by Enthought Traits/TraitsUI. Boilerplate code is removed installing automatic bindings between Model properties and UI widgets, and automatically selecting the appropriate mapping between data types and their UI representation.

The Model is defined through type-enforcing python descriptors.

```
class Person(HasTraits):
   name = Str
   surname = Str
   age = Range(0, 100)
```

and the Visual Proxy is obtained by invoking `edit_traits()`

```
person = Person()
ui = person.edit_traits()
```

which results in a window containing two LineEdit and a Slider.

Data Dialog

Motivation

Data Dialog is a simplified and practical design to retrieve information from the User by means of a modal dialog. It is generally used to retrieve preference information.

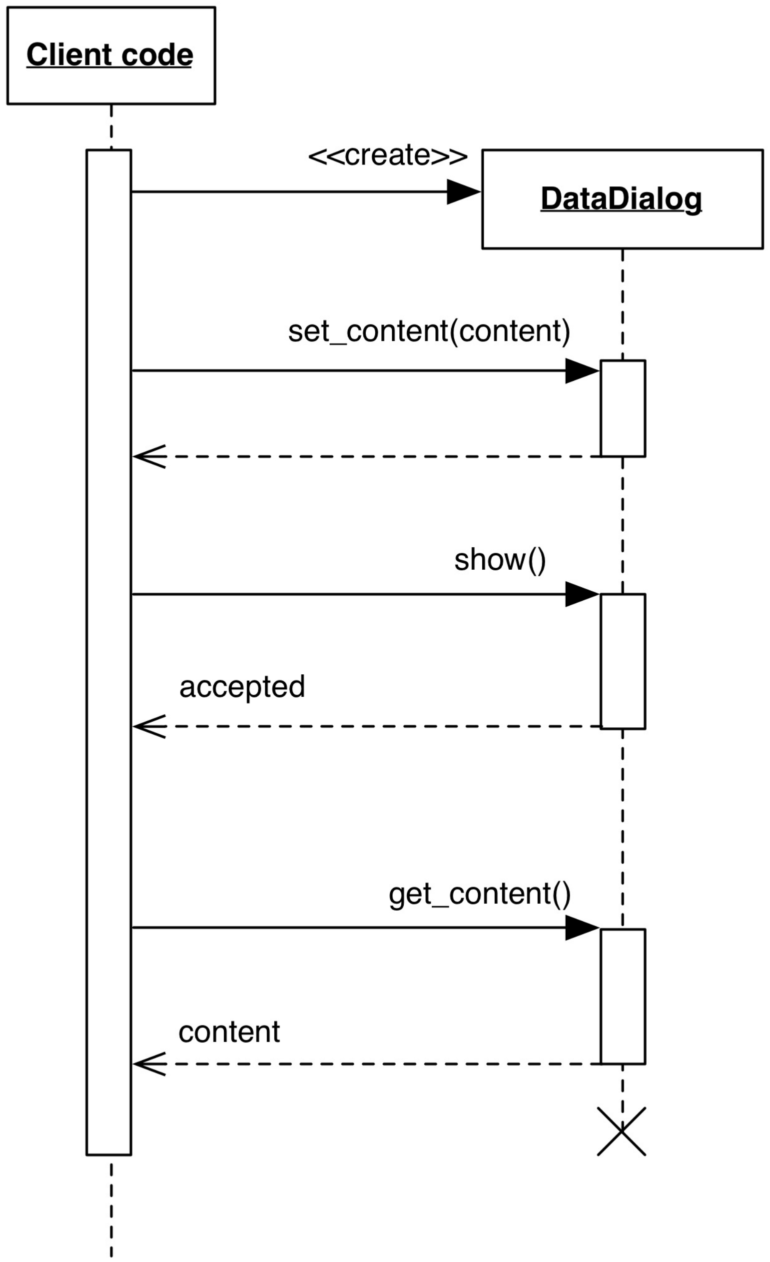
To use a Data Dialog, the following constraints must be respected:

- It must be Modal (i.e. when visible, interaction with the rest of the application is prevented)
- It only allows Accept (`Ok` button) and Reject (`Cancel` button) operations, not Apply
- It does not need to synchronize with the application state while visible.

Data Dialog is different from Local Model. A Local Model is a real Model that is connected to the View through notification, but has simply been copied to preserve its initial state if changes are reverted. Data Dialog, on the other hand, is simply a View with an API to populate or retrieve information from its widgets in a trivial representation.

Design

Data Dialog object is instantiated, and its widgets are populated through an appropriate method call `set_content` . Data is passed in a trivial representation (*e.g.* a properly keyed dictionary) as an argument of this method.



The dialog is then shown to the User. As explained in the Motivation, the `show` operation must block and produce a modal dialog. The User can interact with it and modify the presented values. Basic validation can be performed and reported.

Eventually, the User will issue either an "Ok" or "Cancel". With an "Ok", the new data is gathered from the Dialog through a `get_content` method returning the same trivial representation. The client code will then process this information appropriately. If the User issues a "Cancel", the gathered information is simply discarded.

Testability of the Data Dialog itself is potentially complex due to its synchronous nature. Client code however can replace Data Dialog with a mock honoring the same interface, resulting in easier testability of this part of the application.

Practical Example

The following example with Qt will show the main concept outlining Data Dialog. Both idiomatic python and Qt have been ignored to favor clarity, as usual.

The `DataDialog` class implements a Dialog with textual fields, an Ok and Cancel buttons.

```
class DataDialog(QDialog):
    def __init__(self, parent=None, flags=0):
        super(DataDialog, self).__init__(parent, flags)
        self._line_edits = {}
        self._data_fields = ["name", "surname", "age"]

        layout = QGridLayout()
        for row, field in enumerate(self._data_fields):
            layout.addWidget(QLabel(field), row, 0)

            line_edit = QLineEdit()
            layout.addWidget(line_edit, row, 1)

            self._line_edits[field] = line_edit

        ok = QPushButton("Ok")
        cancel = QPushButton("Cancel")
        self.connect(ok, SIGNAL("clicked()"), self.accept)
        self.connect(cancel, SIGNAL("clicked()"), self.reject)
        layout.addWidget(cancel, len(self._data_fields), 0)
        layout.addWidget(ok, len(self._data_fields), 1)

        self.setLayout(layout)
```

The core of the design resides in the `set_content/get_content` pair: `set_content` accepts a dictionary with appropriate data for the dialog fields, and fills the widgets with its contents. `get_content` retrieves the data from the widgets and returns them as a dictionary to the client code.

```
class DataDialog(QDialog):
    # <...>
    def set_content(self, data):
        for field in self._data_fields:
            line_edit = self._line_edits[field]
            if field in data:
                line_edit.setText(data[field])

    def get_content(self):
        data = {}
        for field in self._data_fields:
            line_edit = self._line_edits[field]
            data[field] = line_edit.text()

        return data
```

The client code interacts with the dialog by setting and retrieving the data through these two methods:

```
data = {"name": "Albert",
        "surname": "Einstein",
        }

data_dialog = DataDialog()
data_dialog.set_content(data)

if data_dialog.exec_() == QDialog.Accepted:
    print("Dialog Accepted. Content:")
    print(data_dialog.get_content())
else:
    print("Dialog Canceled.")
```

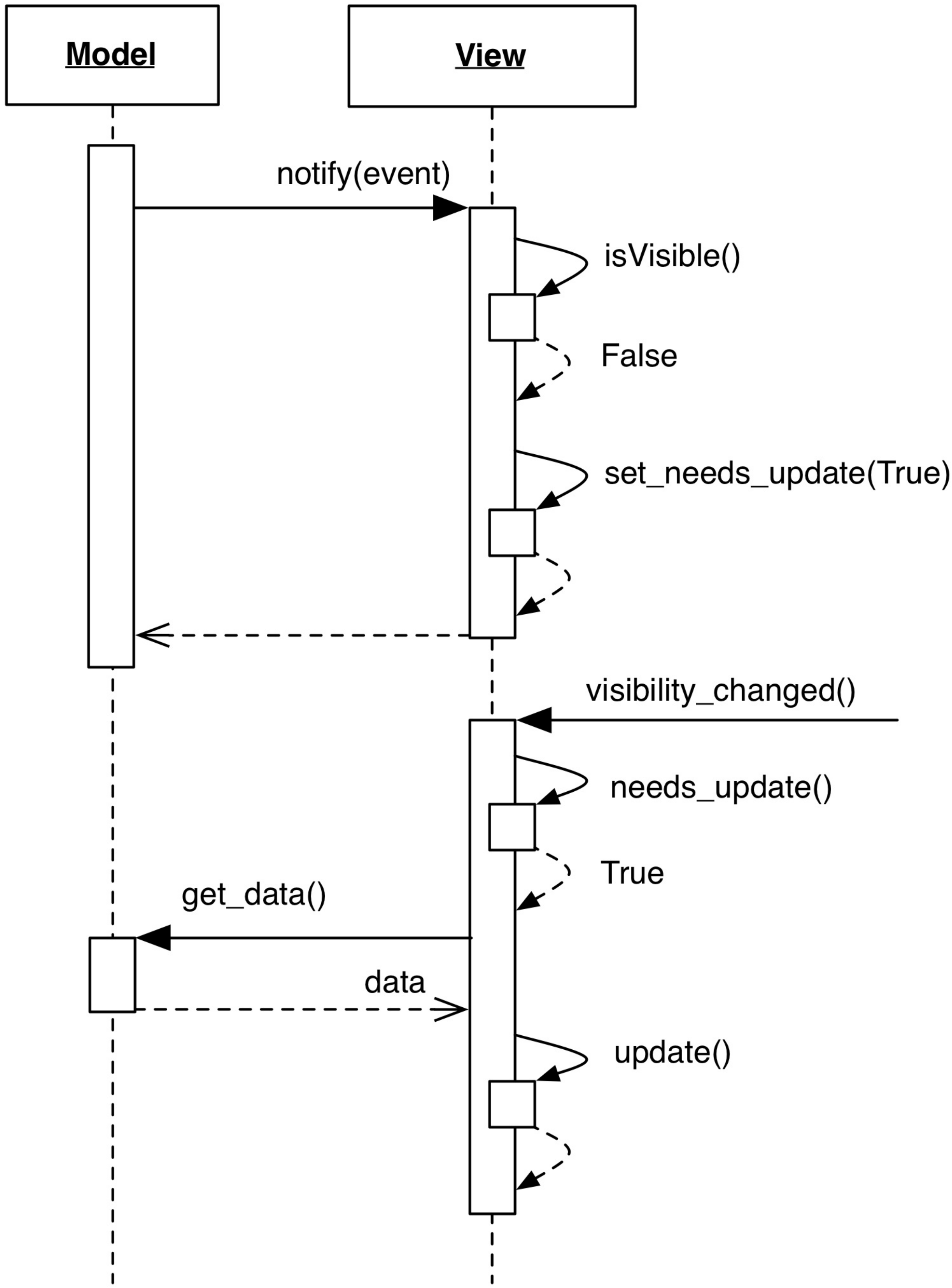
Visibility Allowed Notification

Motivation

Views that are hidden to the User generally do not need to react to notifications from the Model: depending on the nature of the View and the optimizations of the UI toolkit used, this can result in performance degradation from mild to severe. Machine cycles can be saved by ignoring the notification altogether, according to the visibility status.

One possible approach to this optimization is to unsubscribe the View from the Model when hidden, and re-subscribe when shown. This option requires some bookkeeping, and is generally less appealing than the alternative: to simply interrupt further processing of the notification when the View is hidden. When the View becomes visible again, synchronization with the Model must occur, but only if an actual change has taken place. Failure to do so would slow down the return of the View without reason.

Design



When a notification is delivered to the View, the View checks for its visibility. If not visible, it simply sets a `needs_update` flag.

When the View is made visible again, and assuming the View preserves its visual state even when hidden, the `needs_update` flag is checked. If the flag is set, the View resynchronizes against the Model contents, otherwise, it just presents the old visual appearance.

Practical Example

Implementation of this feature is trivial:

```
def notify(self):
    if not self.isVisible():
        self.needs_update = True
        return

    self.refresh_from_model()

def showEvent(event):
    if self.needs_update:
        self.refresh_from_model()
        self.needs_update = False

    super().showEvent(event)
```

We resync against the Model in `refresh_from_model`. This method is called at `showEvent()`, the Qt method for handling widget show events. We save an additional resync against the Model by explicitly checking for the `needs_update` flag.

The `notify()` method called by the Model checks visibility first and sets the `needs_update` flag if the window is not visible, saving a `refresh_from_model` call.

Synchronization of view state

in some cases, we have the following scenario. The View retains more state than the model. If the model changes, and the view just fetches the new model data and repopulates itself, the view will lose its state. This can be an annoyance.

An example is given by a View displaying a file tree, with the model providing filesystem data. The view holds state in the form of opened/closed tree branches, or the position along the scrolling area.

If a new file is added, the model will report a change has occurred, triggering a View actions. A naive solution of purging and updating the content will have poor usability consequences: the view will remove all its content, and reinsert the new data from the model. This is unacceptable not only for performance reasons (although good caching may solve it), but also because the View state will be discarded. Opened tree branches will be reset, and the scrollbar will reset. Additionally, some flickering may appear of the View deleting all its entries and repopulating itself again.

Variations on the Controller

- Supervising controller
- Event Filter: Examine and potentially handle UI events before they reach the View.
- Mediating Controller: Connects widget and properties.
- Coordinating controller: Controllers managing the lifetime of the application, rather than a specific View.
- Action: Controller with visual representation encapsulating a command.
- Action Filter: Perform behavior before or after the execution of a Controller action.

Supervising Controller

Motivation

Toolkits and UI frameworks can provide declarative generation of a View with direct binding between data in the Model and a corresponding Widget in the View. (see Variations on the Triad: Data Binding). This declarative approach simplifies development, but can be of limited expressive power.

Supervising Controller complements the declarative View/Model connection to handle complex logic that is beyond the capabilities of the Data Binding setup.

Design

The overall design has two channels for modifying the View:

- The first channel connects the Model to the View through declarative Data Binding. When a specific property in the Model changes, the declared associated View's Widget is updated to the new value.
- The second channel is responsible for those modifications that cannot be handled easily by the declarative mechanism. For example, the Supervising Controller receives the notification of the property change from the Model, and acts on the View to change the background color of the Widget, for example to report an incorrect value.

The Controller channel thus enhances the Data Binding channel by acting on the View. The same mechanism handles the opposite communication route (View to Model): direct Data Binding handle transporting simple data from the View's widgets to the associated Model's properties, but more complex events, or events requiring complex Model transformations, or events that are not mappable to a Model property (like, for example, a button press) are handled by the Supervising Controller.

References

- Martin Fowler - Supervising Presenter

Event Filter

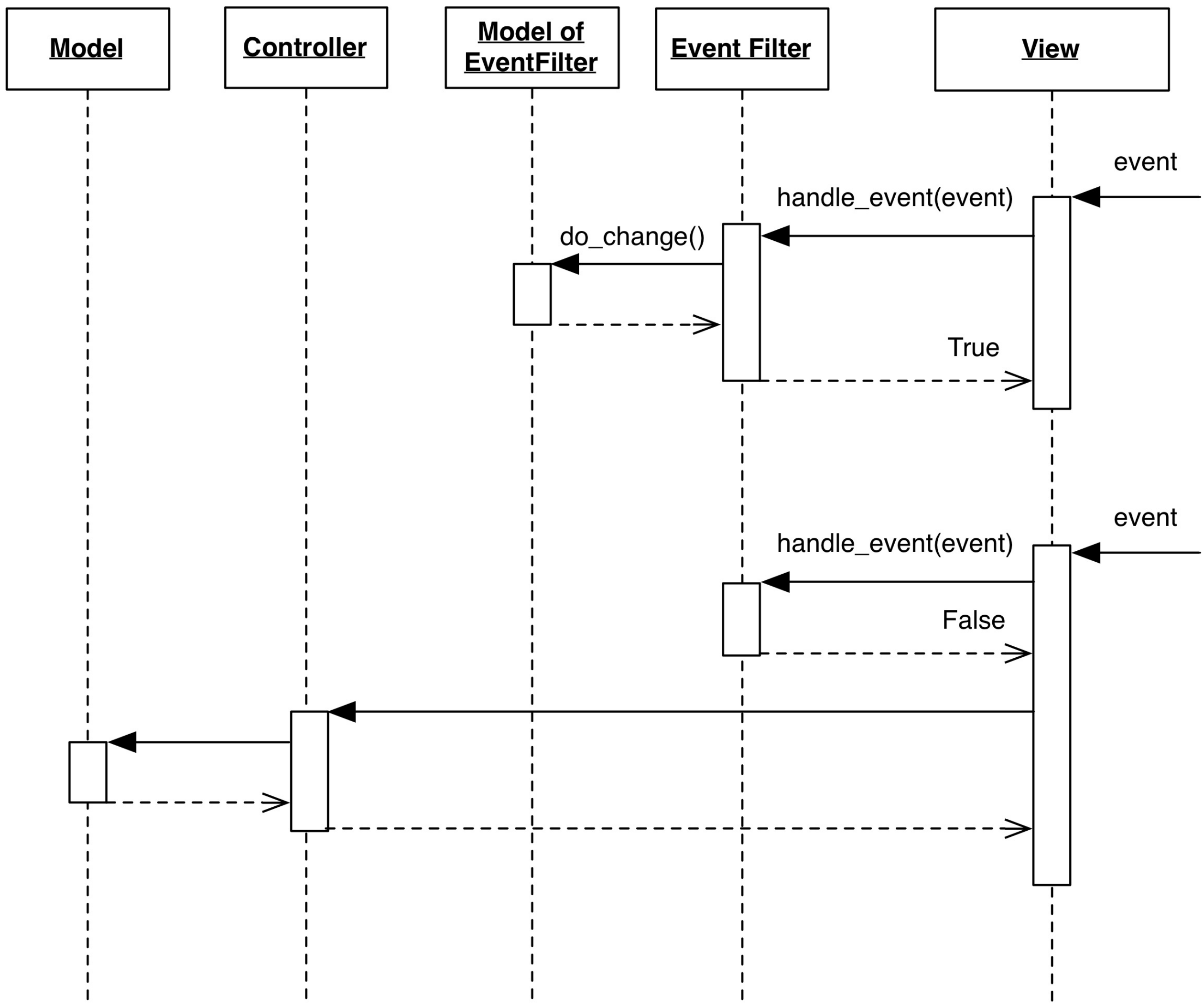
Motivation

An Event Filter is a special Controller that encapsulates redirection of the UI event flow: it intercepts UI events before they are delivered to a specific View. Once intercepted, events are interpreted and handled by the Event Filter, which can perform specific actions or suppress the events completely so that they never reach the View.

Design

Event Filters can be designed either as objects with a specific interface, or as function objects with a signature accepting the event. They are generally designed to be pluggable at runtime through an appropriate interface on the View.

Before being handled by the View, a UI event is dispatched to the View's registered Event Filter, if present. The Event Filter is free to respond to the event by acting on Models, Views or Controller objects which may or may not be known by the View. Once the Event Filter completes its execution, it communicates to the View if the event should be processed as usual or not. In the latter case, the View acts as if the event never occurred.



From the interaction diagram:

1. Events are routed into the Filter by the View, through a `handle_event(event)` interface
2. The Event Filter can now recognize a specific event and perform operations on the Model it is connected to.
3. If the Event Filter handles the event, The View is notified through either a flag on the event class or a boolean return value for the `handle_event()` method. In the case presented above, the latter strategy is used. The `True` return value implies that the event was handled by the Filter and should not be handled further by the View.
4. If `EventFilter.handle_event()` returns `False`, the View handles the event as usual. Note that the Event Filter might choose to handle the event and still return `False`, allowing the View to operate as usual.

As a consequence of its design and purpose, the Event Filter must be dependent on the UI, and must know how to handle the UI Event interface.

Practical Example

The Qt toolkit offers a clear example of an Event Filter. To define an `EventFilter` class, it is sufficient to use `QObject` as a base class, implementing the method `QObject.eventFilter()`. This method will be responsible for the processing of the events. An instance of `EventFilter` is then installed onto a target object through `installEventFilter()`. Qt implementation is very flexible, and allows any object to hold event filters, not only UI components.

The following program illustrates the concept

```
import sys
from PySide import QtGui, QtCore
app = QtGui.QApplication(sys.argv)

class EventFilter(QtCore.QObject):
    def eventFilter(self, receiver, event):
        print(type(receiver), type(event))
        return False

event_filter = EventFilter()
button = QtGui.QPushButton("hello")
button.installEventFilter(event_filter)
button.show()

app.exec_()
```

The example creates a `QPushButton` widget and installs an `EventFilter` instance on it. UI events (mouse movements and clicks, key presses, show/hide, resize, repaint etc.) that are meant for the `QPushButton` are first dispatched to `EventFilter.eventFilter()`, then to the `QPushButton` if and only if the `eventFilter` method returns `False`.

Mediating controller

Motivation

Connects one specific widget with one specific property. Very reusable. Non-declarative form of data binding. Can do validation.

FIXME: We should already have it somewhere in the notes or the coarse vs. fine view.

Practical Examples

NSController and Cocoa bindings.

References

- Mediating Controllers in OSX

Coordinating controller

Motivation

Overall controller that oversees the application, its life cycle, other controllers, setting up connections and so on.

Action

Motivation

User interfaces often require a given operation to be available from different points of access: the menubar, contextual menus, toolbars, and keyboard accelerators. An effective design choice to handle this use case is to define an **Action**, a pluggable entity encapsulating Command behavior and visual/UI related information, such as its icon, tooltip, keyboard accelerator, enabled status and so on.

Design

An Action is peculiar in design characteristic, combining design features of Command, Controller, and Model.

Like a Command, an Action normally provides an abstract method `triggered()` to reimplement with the required behavior. Alternative strategies not requiring subclassing allow to register a callback that is executed when the action is triggered. Differently from a Command, the Action is more liberal in performing UI operations, such as showing a dialog, while Commands generally act exclusively on Models.

Actions are also both Controllers and visual Models: different Views accept the same Action and visually represent it in a different way. For example, a MenuBar might represent it as an icon followed by a title, a ToolBar might display just the icon and a tooltip, and the application as a whole might not represent it visually, but activate it when its keyboard shortcut accelerator is invoked. Information about the visual aspect of the Action are contained in the Action itself, fulfilling a Model-like role for this information.

Views supporting Actions are generally a form of visual container (e.g. Menubar, Toolbar) and provide an interface to add and remove them.

Practical Example

Qt supports a regular example of Action with the QAction class. QAction exposes a `triggered` Signal that is emitted when the Action is activated. This Signal can then connected to any slot callback, where actual Controller behavior takes place.

```
menubar = self.menuBar()
toolbar = self.addToolBar('Toolbar')
file_menu = menubar.addMenu('&File')

quit_action = QAction(QtGui.QIcon('quit.png'), '&Quit', self)
quit_action.setShortcut('Ctrl+Q')
quit_action.triggered.connect(qApp.quit)

file_menu.addAction(quit_action)
toolbar.addAction(quit_action)
```

In this example, we created an Action for quitting the application, then add it to both the File menu entry and the toolbar. Clicking on either entry, or using the Ctrl+Q accelerator, will invoke qApp.quit() and quit the application.

Variations on the Triad

- Model View Adapter
- Model GUI Mediator
- Application Controller
- Push vs. Pull
- Reenskaug MVC
- Dolphin MVP: Rationalize Application model into a more active role.
- Presenter First
- Taligent MVP: Defines a broad set of protagonists to define the most common needs of a generic application.
- Presenter Adapter View
- Model View Viewmodel
- View Controller View
- Commands
- Visual Editor
- Command Notification
- Qt MVC
- Supervising Presenter
- Presentation Model
- Data Binding

Model-View-Adapter (MVA, Mediated MVC, Model-Mediator-View)

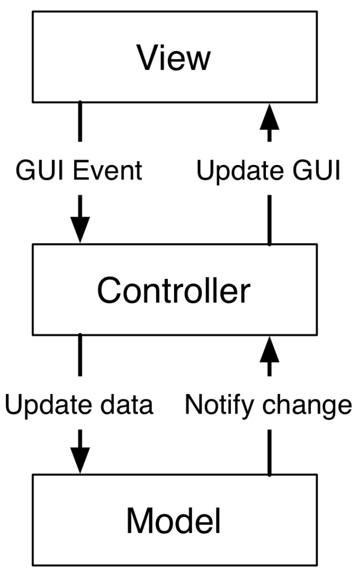
Motivation

Model-View-Adapter is a variation of the Triad where all communication between Model and View must flow through a Controller, instead of interacting directly as in a Traditional MVC Triad. The Controller becomes a communication hub, accepting change notifications from Model objects and UI events from the View.

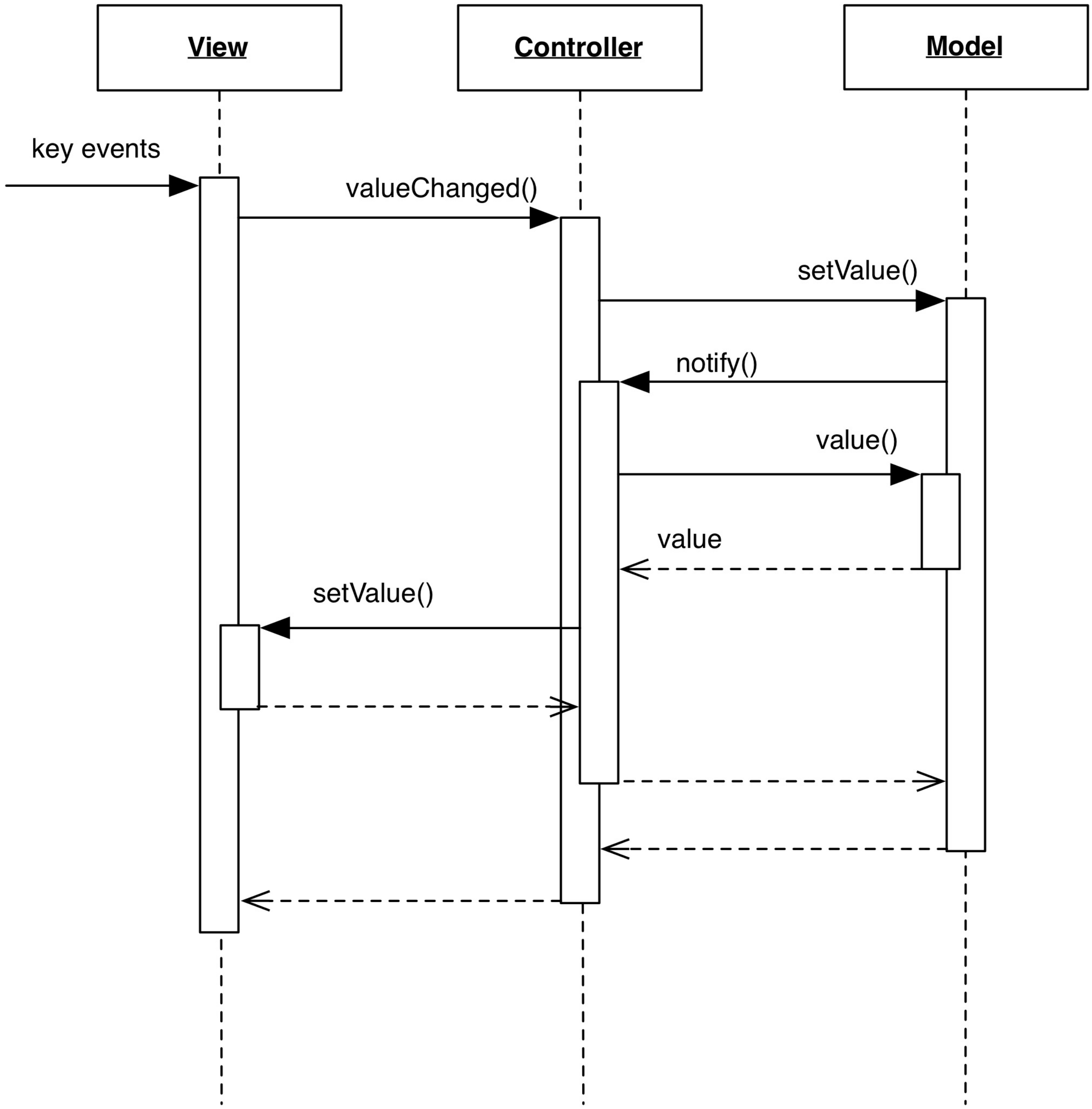
This approach might appear excessively strict, but has some advantages: the communication network is artificially constrained, making it easier to evaluate and debug. All action happens in the Controller, and the View can be created from off-the-shelf widgets without any Model-specific variation.

Design

MVA is an implementation of the Mediator pattern. Controllers are generally referred as Adapters or Mediators. The Model and the View do not hold references to each other, they do not exchange data nor interact directly.



The pattern of communication in MVA can be represented with the following interaction diagram



which can be described as

1. The View receives a UI event. It calls an appropriate method on the Controller.
2. The Controller sets the value on the Model.
3. The Model notifies its listeners of the change, among which is the Controller itself. As already pointed out, in a MVC approach this notification would be sent to the View.
4. The notified Controller fetches information from the Model and updates the View.

With the Controller in full control on the dialog between the two remaining parties, smart tricks can be performed on the “in transit” data: for example, the Controller could be responsible for formatting, translating or ordering the data from the Model.

Practical Example 1: With Subclassing of the Views

common in Apple OSX Cocoa Framework.

We examine a MVA implementation of the Engine example in two steps. In the first step, we will keep subclassing the Views, to present the general concept. In the second step, we will remove the need for subclassing the Views.

The Model is unchanged: it stores rotations per minute information and notifies about changes

```
class Engine(BaseModel):
    def __init__(self):
        super(Engine, self).__init__()
        self._rpm = 0

    def setRpm(self, rpm):
        if rpm < 0:
            raise ValueError("Invalid rpm value")

        if rpm != self._rpm:
            self._rpm = rpm
            self._notifyListeners()

    def rpm(self):
        return self._rpm
```

The two View classes, `Dial` and `Slider`, are now unaware of the Model. Instead, they know about the Controller, and accept changes to their content through the `setRpmValue()` method. A matter of taste can decide the semantic level of this method. Should it talk “domain language” (i.e. Rpm) or not (i.e. the method should just be named setValue). In any case, Views behave differently with respect to the issued value (the Slider needs its value scaled appropriately). One could handle this difference in the Controller, but this will be considered in the next step.

When the user interacts with the `Dial`, the Controller `changeRpm()` method is directly invoked, in this case via the Qt Signal/Slot mechanism

```
class Dial(QtGui.QDial):
    def __init__(self, *args, **kwargs):
        super(Dial, self).__init__(*args, **kwargs)
        self._controller = None
        self.setRange(0, 10000)

    def setRpmValue(self, rpm_value):
        self.setValue(rpm_value)

    def setController(self, controller):
        self._controller = controller
        self.connect(self, QtCore.SIGNAL("valueChanged(int)"),
                           self._controller.changeRpm)
```

For the `Slider`, the interface is the same, but the internal implementation is different. Again, the `setRpmValue` allows the Controller to change the View contents. In this case however, a proper transformation of the data is performed to deal with the specifics of the `Slider` behavior, whose range is from 0 to 10. Similarly, when the User interact with the `Slider`, the method `_valueChanged` will be invoked, which in turn will issue a call to the Controller `changeRpm()` method, after transformation of the parameter

```
class Slider(QtGui.QSlider):
    def __init__(self, *args, **kwargs):
        super(Slider, self).__init__(*args, **kwargs)
        self._controller = None
        self.connect(self, QtCore.SIGNAL("valueChanged(int)"),
                           self._valueChanged)
        self.setRange(0,10)

    def setRpmValue(self, rpm_value):
        self.setValue(rpm_value/1000)

    def setController(self, controller):
        self._controller = controller

    def _valueChanged(self, value):
        if self._controller:
            self._controller.changeRpm(value*1000)
```

The Controller class handles the Model and the two Views accordingly. It registers for notifications on the Model, and it receives notification from the Views on its `changeRpm()` method, where it modifies the contents of the Model. When the Model communicates a change, it pushes the new value to the Views

```
class Controller(object):
   def __init__(self):
       self._views = []
       self._model = None

   def setModel(self, model):
       self._model = model
       model.register(self)

   def addView(self, view):
       view.setController(self)
       self._views.append(view)

   def changeRpm(self, rpm):
       if self._model:
           self._model.setRpm(rpm)

   def notify(self):
       for view in self._views:
           view.setRpmValue(self._model.rpm())
```

Practical Example 2: With generic Views

In the previous example, removing the dependency of the Views on the Model did not allow us to use generic Views. In this example we will move the value translation logic from the Views to the Controller, thus removing the need for subclassing the Views, an important advantage of MVA.

```
class Controller(object):
   def __init__(self, model, slider, dial):
       self._slider = slider
       self._dial = dial
       self.connect(self._slider, QtCore.SIGNAL("valueChanged(int)"),
                          self._sliderChanged)
       self.connect(self._dial, QtCore.SIGNAL("valueChanged(int)"),
                          self._dialChanged)
       self._model = model
       model.register(self)

   def changeRpm(self, rpm):
       if self._model:
           self._model.setRpm(rpm)

   def notify(self):
       for view in self._views:
           view.setRpmValue(self._model.rpm())
```

Practical Example 3: With forwarding of Model events

If the Model emits qualified events, the Controller could simply forward them to the View.

Model-GUI-Mediator

One problem with Model-View-Adapter is that it assumes Views are derived classes, each implementing specific behavior.

In the previous example, each View performed a specific transformation to the data before displaying: the Dial left it as is, while the Slider divided it by 1000.

In the Model-GUI-Mediator, the desire is not to subclass the toolkit's widgets, because it generally leads to proliferation of View classes. Instead, widgets are used as they are, off-the-shelf from the toolkit. The obvious consequence is that logic that is pertinent to the conversion of data for visualization must go somewhere else. The Controller seems the obvious choice, however keeping the same design as in MVA would be cumbersome: the single Controller would have to differentiate the Views, and submit properly transformed data to each View. A better solution is to have different Controllers, one per each View, doing the relevant transformation. The code would therefore be like the following: The View being an off-the-shelf component means it does not know anything about the Controller. All the signal setup is done by the individual Controllers. Also, off-the-shelf classes are not implementing the Observer pattern

```
class DialController(object):
   def __init__(self):
       self._view = None
       self._model = None

   def setModel(self, model):
       self._model = model
       self._model.register(self)

   def setView(self, view):
       self._view = view
       self._view.setRange(0,10000)
       self._view.connect(self._view,
                          QtCore.SIGNAL("valueChanged(int)"),
                          self.changeRpm)

   def changeRpm(self, rpm):
       if self._model:
           self._model.setRpm(rpm)

   def notify(self):
       if self._view:
           self._view.setValue(self._model.rpm())
```

And for the Slider it would be

```
class SliderController(object):
   def __init__(self):
       self._view = None
       self._model = None

   def setModel(self, model):
       self._model = model
       self._model.register(self)

   def setView(self, view):
       self._view = view
       self._view.setRange(0,10)
       self._view.connect(self._view,
                          QtCore.SIGNAL("valueChanged(int)"),
                          self.changeRpm)

   def changeRpm(self, rpm):
       if self._model:
           self._model.setRpm(rpm*1000)

   def notify(self):
       self._view.setValue(self._model.rpm()/1000)
```

The setup now can simply make use of off-the-shelf QDial and QSlider instances

```
dial = QtGui.QDial(container)
dial_controller = DialController()
dial_controller.setView(dial)
dial_controller.setModel(engine)

slider = QtGui.QSlider(container)
slider_controller = SliderController()
slider_controller.setView(slider)
slider_controller.setModel(engine)
```

The Model-GUI-Mediator approach basically has the Controller adapt the off-the-shelf widget to be aware of the Model. This requires no subclassing. In a sense, Model-GUI-Mediator is similar to Document-View, but it reorganizes competences in a different way and splits the View into off-the-shelf functionality and application-contextual functionality.

Application Controller

Push vs. pull

Defining “push-vs-pull” within the realm of MVC can lead to confusion, because it is an overloaded term. We will talk here about the We will discuss of the “Push-pull” model for MVC also in the context of web frameworks.

This is known as the “pull” model. The view is informed of its outdated state, and pulls information from the model. The alternative is the “push” model, where the model notifies the view and passes its new state to the view.

Push model: the view register with the model and receives notification of changes. Pull model: the view fetches the new state from the model.

Note: unclear. Also possible mixup between overloaded meaning in GUI MVC and Web MVC.

Reenskaug MVC

Motivation

Trygve Reenskaug formulated MVC first in 1979. His original approach is different from modern forms of MVC, and for all purposes obsolete. It is presented here to explain the historical formulation of MVC and its motivations.

Design

The best way to present Reenskaug MVC is to compare it against a Traditional MVC design. Like in Traditional MVC, Reenskaug MVC has:

- A Model representing knowledge about our data.
- The View visually represents the Model, obtaining information by invoking its methods.
- The Controller has UI event handling duties, but for different reasons.

Crucial differences exist in the Controller, the View, and in a specialized object absent from Traditional MVC, the Editor:

- Reenskaug Controller handles UI visual layouting and UI primary events, converting these events into operations on the View. In Traditional MVC, this is done by the View.
- Reenskaug View directly modifies the Model through Model methods calls. It does so with the assistance of an Editor. In Traditional MVC, this is done by the Controller.

This design is a consequence of the technical environment of the time: Views' widgets were simple renderings on the screen, with no functionality to receive and process events from input devices. This task was assigned to Controllers. At any given time, only one Controller was considered active and would receive UI events from the event loop. Controllers were organized in a hierarchy and had to negotiate the active status among themselves in response to UI events and the expectations of the User/UI interaction. If a Controller found itself not authoritative to handle a specific event, it would delegate to the Controller above in the hierarchy.

With the Controller performing layout/event handling duties, the responsibility for Model modification was handled through an additional player, the Editor, with cross-functional demands and dependencies.

An Editor is brought into existence on demand: the Controller asks the View for an Editor, which is presented to the User. UI events from the Controller are routed to the Editor, which converts these events in method invocations onto the View's API. Finally, the View modifies

the Model.

Reenskaug Controller

With the introduction of smarter Views able to handle events,

The third difference is the presence of the Editor as a “View-extension helper” that the Controller uses in order to perform its task. The reason for this design is that the Controller must have a View-contextual entity to present to the User. For example, a GUI Label might require a TextEdit field as an Editor, if the text is “free form”, but a ComboBox if the label can only contain discrete values. Only the View part can know its appropriate Editor.

into operations on the View. The View is not supposed to know about primary events.

Since then, widgets gained ability to handle events In other words, most of the task initially assigned to a Reenskaug's Controller are now taken care of by an underlying GUI framework.

Dolphin Model-View-Presenter

The Dolphin Model-View-Presenter (MVP) schema is an evolution of the Application Model approach. Although it is derived from the Taligent/IBM strategy with the same name, we will examine Dolphin first as it is simpler to describe within the concepts we already introduced. The Dolphin strategy is also the one most often referred as "Model View Presenter" without additional clarification. To add to the nomenclature, Fowler identifies the Presenter with the more appropriate "Supervising Controller".

As we introduced in the Application Model section, the main purpose of this Model is to hold visual state, acting as an intermediate between the Domain Model and the View/Controller pair. The Application Model was a model in every respect in terms of design: it performs notifications, remains oblivious of its listeners, is directly accessed by the View and modified by the Controller. Yet, due to the strictly specific nature of the visual state, it would be convenient if the Application Model could handle visual logic and refer to the View directly, like the Controller does, while still keeping and handling View state.

Let's analyze the Controller: with widgets of modern GUI toolkits handling low-level events (e.g. physical keyboard presses), the controller has only the duty of modifying the models according to higher level events (e.g. textlabel content modified). These events are then transformed by Controller logic in actual Model changes, some of which may have an impact on the visual state, which is stored in the Application Model. It seems like a good idea to have an Application Model containing this visual state if the assumption is that this state (e.g. a field being red) is shared among Views. Once again, this state is almost never shared and mostly tied to a specific View.

Summing up, the roundabout mechanism the Controller uses to take care of purely visual state would be considerably simplified if we define a new role, the **Presenter**, which combines the Application Model and the Controller in a single entity.

Like the Application Model, the Presenter:

- holds visual state, and keeps it synchronized against changes in the Domain Model
- converts business rules (e.g. engine rpm number too high) into visual representation (e.g. label becomes red)
- eventually handles state for selection, and application of actions to the subset of the Model specified by this selection.

and like the Controller, the Presenter:

- it is tightly coupled to the View
- refers to the View directly, and can act on it to alter its visual aspect.
- handles events forwarded by the View, converting them into action through proper logic.
- modifies the Domain Model, which contains no visual state
- handles View logic according to the View state it contains

The Domain Model is a strict domain model and is unchanged, and is still accessed by the View for data extraction and from the Presenter for data modification. The View is an aggregation of widget, fetches data directly from the Domain Model, instead of having to rely on the Application Model as a forwarder. The View behavior is now hybrid Active/Passive, fetching Domain data directly from the Domain Model but with visual aspects applied by the Presenter (Passive) directly acting on the widgets. A variant with a fully Passive View is possible, and is known as **Presenter First**.

If the user changes the content of a widget, the presenter is informed, it fetches the value from the widget, and acts on the model. The model is listened to by the view, which updates itself accordingly, and by the presenter itself, which now sets the color.

The view may manipulate its own logic and aspect internally if it does not need to modify the model.

every view has a specific presenter. There's a strict relationship between the two

Each triad is independent and does not know about the other ones. in general, if the triads must coordinate each other, they can rely on an "application model". When the application Model for a given triad is requested to perform a change, it signals its intention to an application coordinator. This coordinator can eventually decide to create a new triad, if needed.

FIXME: Add Picture FIXME: reformulate in general

FIXME: Write something about data transfer objects as a transferring entity of data between the model and the presenter. It is possible to add a service layer between the model and the presenter that is responsible for packing the data from the model into a DTO that the presenter then uses to set the view's contents.

FIXME: The application model approach assumes that there are multiple View/Controllers acting on the same application model, but in practice it's very rare that an application model is generic enough to be applicable to multiple views or controllers, exactly because its state is designed to satisfy the visual needs of a specific View.

FIXME: Add the fact that the controller is talking to the view as a generic object that generates events. This allows easy repleaceability of the view, not only with a mock, but also with a different renderer. As long as it generates the same events and responds to the same

interface. this may seem trivial, but if the difference in interface is large, this replaceability is not achievable.

FIXME: Division of functionality in the presenter must keep into account if the functionality would make the presenter grow too much and act as a data bottleneck. Sometimes if the logic is complex and the view is intrinsically complex, it pays off to let this logic live in the View and have a simple presenter.

FIXME: In a large application, a presenter normally handles only one view, but it could handle multiple views if they are closely related.

FIXME: Communication choices. 1) View holds reference to the Presenter. View invokes directly methods on the presenter, instead of sending events. or 2) View emits events. Presenter listens

Presenter holds a reference to the View. directly invokes its methods.

In the original MVC, the controller was mainly responsible for handling user events. in MVP, the Presenter is responsible for handling Model changes.

Composed of supervising controller, passive view.

Moves presentation logic from the Model (as in aplication model) to the presenter.

MVC vs. MVP:

- Controller in MVC: main objective is to handle user events. Model modification is a consequence of it. With smart widgets, user events are handled by the widgets, and then forwarded to the controller. MVP: Presenter main objetive is updating the model, and keeping the visual state (as in application model) handling events passed by the view is a consequence of it.

The presenter can be instantiated either by the client code, or directly by the view. If this is the case, the View must know the model, so that it can instantiate the Presenter and pass both the model and itself to it.

Concerning callability on sone side or the other (e.g. event vs. method call)

write that the view has two ways to interact with the controller and forward events: strong coupling through direct invocation, or loose coupling through raising events at a higher semantic level. forwarding is done depends on the degree of coupling you allow between the View and the Presenter. If the view must invoke directly a Presenter's method, obviously it must know its interface, so it must hold a reference to it and know its interface. The alternative is that the view is oblivious to who is listening, and just broadcasts events (commands) to report the button press. The presenter observes these events and take

appropriate action when triggered. As you can see, the difference is subtle, and apparently irrelevant, but it can be useful depending on the degree of coupling and self-containment of the view vs. the controller (Presenter)

Presenter First

More than a design, **Presenter First** is a discipline for coding a Dolphin MVP. As briefly stated in the closing remarks of the previous section, it is possible to deploy a Dolphin MVP variation where the View is fully passive.

The presenter performs calls on both the model and the new, and receives events from either of them.

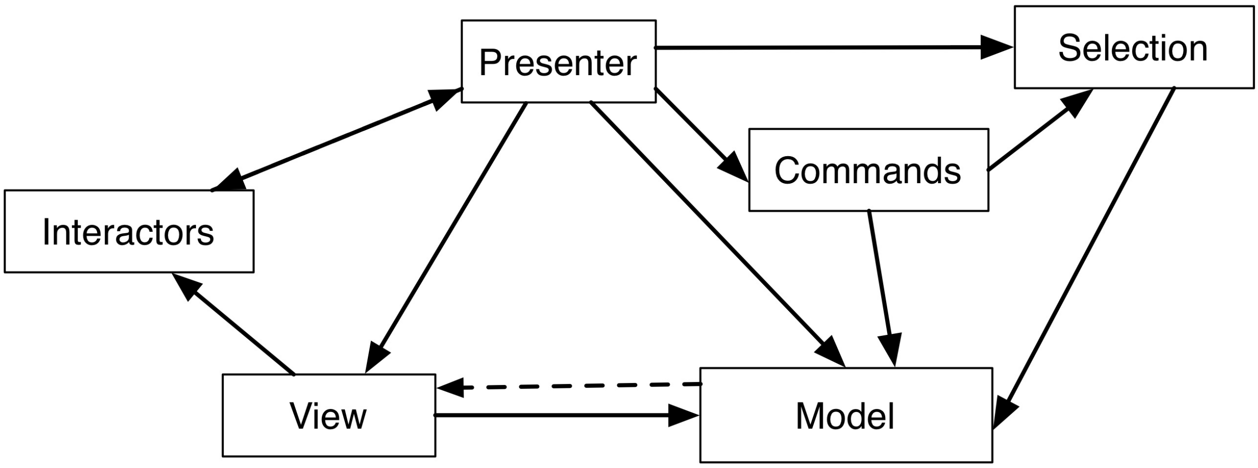
Presenter First takes advantage of this choice to focus on the Presenter as the starting point for development of a specific triad. The idea is that, by creating the Presenter under direction of the customer's desires, there's a chance to progressively discover the required interfaces of both the View and the Model by coding against mocks and lastly implementing the discovered interfaces.

The point of presenter first is to code the presenter to satisfy customer needs, and code against it first. While the model provides a functionality and data oriented access, the presenter describes the actions in terms the user understands or specifies. The presenter has no state. It relies on the view and on the model to hold state, performing operations on these two. In principle, the Presenter should listen to events from either the model or the view, meaning that it should not need public methods. All the connections between events and presenter methods are setup by the presenter itself.

The Presenter should be tested strenously. The View, being a Humble View, can be left untested.

Taligent/IBM Model-View-Presenter (MVP)

Until now, we saw several strategies to address modern requirements such as undo/redo, selection, and View-related state. Taligent, a subsidiary company of IBM, formalized these strategies into the so-called Taligent Model-View-Presenter architecture. An equally named, but different strategy is the Dolphin Model-View-Presenter, which will be introduced later. At first, MVP seems complex, but in reality is a little step from what already introduced in the previous sections.



The aim is to divide responsibilities in simple, testable entities while moving all logic away from the part that is most difficult to test, which is the View.

Taligent MVP organizes entities so that they can be divided into three classes: A data oriented class, a GUI oriented class, and the Presenter. The Data oriented class is composed of the following parts:

- A **Model**, whose role is purely of business.
- A **Selection**, holding information about the subset of the Model that will be affected by user's actions
- A set of **Commands**, encapsulating operations that can be performed on the Model according to the Selection, and supporting undo/redo semantics.

The GUI oriented class is instead composed of

- a View, representing the content of the Model, and accessing the Model directly.
- Interactors, mapping UI events at either low (mouse click, keyboard press) or high level (menu entry choice, click and drag selection) into actual commands to execute on the model.

Finally, the Presenter class only contains the Presenter, which is an overarching director object orchestrating the interaction of the above objects. Generally, there's a Presenter for every View.

Having this structure has the advantage of distinguishing the range of application of a given operation vs. the operation itself, through the Selection/Command distinction. This simplifies the code, and solves notification issues so that notification is sent only when the full operation is performed on all members of the selection.

Note that the MVP selection is not a Selection Model. It describes the selected items in the Model on which a command is applied, but the visual representation of this selection is in charge of the View. The Presenter is in charge of obtaining that representation from the View, convert it into a MVP selection, and operate the commands with it. This results in the impossibility to share the visual Selection among Presenters.

FIXME excessive confusion and overload of the terms "class", "object", "entity"

Presenter as a coordinator of the process. One presenter per each view.

FIXME: widgets including both view and controller role. Presenter performs the active role in the model modification. presenter is basically an application model.

chopped up the old MVC roles and reassigned. It seems like MVC, but its objects aggregate different responsibilities.

Presenter-Adapter-View

Sometimes, in the management of the visual aspect of the view, it pays off to extract some of the code of the presenter and isolate it into a separate adapter object. This adapter object encapsulates the logic for some visual representation. The presenter, instead of talking directly to the View, talks to the Adapter that transforms this request into a proper rendering on the view.

For example, if we want to color a label red when a value is too high, the presenter can use an adapter where the logic for setting something to red is according to the value. the presenter sets the value on the adapter, which in turns sets the value on the view and changes the color of the label appropriately.

The adapter needs not to know the business rules. The adapter can query the model to ask if a given value is above or below a certain range, and choose the color appropriately.

Once again, the adaptor is simple to test.

In addition, one can extract an "adapter model" and have the presenter push data into the adapter model. The adapter then takes these data, and push them to the specific part of the view.

Model-View-ViewModel (Model-View-Binder)

The MVVM is a specialization of the Presentation Model. It is rather popular in the Windows world, particularly WPF and Silverlight.

MVVM has a traditional model, an active view (generally declared as a XAML description) that handles its own events internally and acts both on the Model and the ViewModel. The View and the ViewModel contents are bound together in a direct simple relationship through bindings. A checkbox on the view can be bound to a boolean field in the ViewModel. In other words, the ViewModel is the “Model of the View” intended for the representation the view has of the data. The Model, in fact, might contain a different representation of the values (for example, in the Model vision of things, that checkbox could represent the existence of a reference between two Model objects). The ViewModel is responsible of mapping its state (the boolean) to setting the reference, and vice-versa.

Similar to Presentation Model/Application Model, however, the ViewModel has no explicit reference to the View, nor explicit code to direct the View. Instead, it uses data binding, which offloads the sync task from the developer.

View-Controller-View

A View-Controller-View is basically a Model-Controller-View where one of the Views is playing the part of the Model for a specific interaction. This occurs when a View must interact with another View to orchestrate its behavior. A simple practical example is a Dialog for a Search functionality, and an Editor providing methods for this functionality. The two Views must interact so that when the user clicks on the “Search” button of the Dialog, the Editor is directed to perform the search.

Commands

Graphical interfaces generally provide Undo/Redo capabilities. This is generally and easily implemented with the Command pattern. The controller, instead of directly performing an operation on the Model, will instantiate a Command object out of a palette of possible Commands. The command object will be instantiated by passing a reference to the Model. This object will normally have two methods execute(), and undo(). The Controller will instantiate the command, and submit it to a Tracking object. The tracking object will call execute() on the Command, and immediately push it into a stack. The Tracking will have two stacks, one for undo, and the other for redo. When the user selects undo, the Tracker will be requested to pop one command from the undo stack, call its redo() method, and push the command in the redo stack. Redo can be implemented by undoing the actual process, or by storing the old state and reverting it. The memento pattern is here useful to save the state of the Model before modification, but of course it can be demanding in memory occupation.

Using the command pattern to modify the model. The model can be a factory for the commands. The command can perform notification of the listeners instead of the model. Another form of qualification: the model forwards the command after execution to the View. Views can analyze the command to respons appropriately.

a command class normally has execute() and undo() methods. it's a functor.

execute does a given action. undo restores the state as it was before. undo can be done either by algorithmic rollback, or by just restoring a memento saved at execute time.

The parameters are defined at instantiation time. execute accepts no parameters but it may return a state (success, failure). This state will be needed to decide what to do with the executed command (add to the undo queue or not).

two stacks: undo queue and redo queue.

execute() push into undo

at undo: pop from undo command.undo() push into redo

at redo pop from redo command.execute() push into undo

Association of the command to the model: the model defines and offers creation of commands. The command can also acts as a notification agent (e.g. syncs the view) instead of the model and can also act as a change object.

FIXME Command is a general design pattern, we will present it here in the context of a MVC need.

Visual Editor

Visual editor, or Qt delegate, is a factory class to create an editor for a proper editable entity.

Command notifications

Notification performed by the altering entities (commands) similar to passive model, where the altering entity is the plain controller.

Nokia Qt

Qt provides Views and associated Models, who are either tabular or hierarchical in nature. The framework also provides derived classes for Views, called Widgets. Widgets combine View and Model in a single class, allowing to store data directly into the view. This approach loses flexibility and ease of reuse of the data contents, but it can be convenient for some specific cases (for example, if you want to control addition and removal directly on the widget). Qt has delegates, that are associated to views. Delegates are responsible for handling controller tasks, and in addition control rendering and editing of these views. A delegate renders data into the view with the paint() method, and creates editors for the data with createEditor(). Default Delegates are installed on every view. The model contains data classified in roles. Some roles are purely data oriented, while other roles (FontRole) are view-level information. The model is therefore responsible for influencing the visual appearance of thje view through the Role mechanism. Of course, this mechanism can also be implemented by a specialized delegate who translates the Data semantic into visual semantic.

FIXME: For example, the QListView is a MV counterpart of QListWidget. There's no controller in Qt, it doesn't need any. In practice, Qt is a Model Delegate Editor design, except that the Delegate is called View, and the Editor is called Delegate. Such is life.

MVC model: modification through slots. Notification via signals.

Emitting before changing the data in the model, to track changes. But careful if the calling code is in another thread.

Qt model sort of a value model, but provides both visual and business data. The data structure is flexible to accommodate tabular and tree data on the same model structure.

Specialized models provide access to specific data (e.g. sql table, filesystem directory). Views are oblivious of the data provider as long as it fulfills the AbstractModel interface.

```
directory = QDirModel()
table = QTreeView()
tree.setModel(directory)
tree.setRootIndex(directory.index("/")
```

Model indexes are created on the model, acting as a factory. Indexes can mutate as new elements are added or removed. A Persistent index is available to refer to a specific entry regardless of modifications to the model.

Model defines row, column and parent. Views don't necessarily use all the degrees of freedom. For example, a table may not need the parent, or a listView may not need the column, but the treeview needs all of them

The view requests only the data that it needs to show. This can save considerable time, and allows the model to do caching strategies or pre-fetching.

Qt also provides sort/filter (Pipe).

Model is a Presentation model.

Complex application models need to be adapted into Qt models if needed. In a sense, the Qt model is an adaptor to a data representation (both visual and business) that is appreciated by the Views provided by qt. With more complex views, or models, the Qt mechanism breaks down, because it's finalized at a very specific data organization and visualization mechanism.

Provides the following Models

QStringListModel: Stores a list of strings QStandardItemModel: Stores arbitrary hierarchical items QFileSystemModel: Encapsulate the local file system QSqlQueryModel: Encapsulate an SQL result set QSqlTableModel: Encapsulates an SQL table QSqlRelationalTableModel: Encapsulates an SQL table with foreign keys QSortFilterProxyModel: Sorts and/or filters another model

QAbstractItemModel

and the following Views

QColumnView, QHeaderView, QListView, QTableView, and QTreeView. QAbstractItemView

The Qt MVC is simple and aimed at specific data representation and visualization. It's not a general framework for MVC.

Delegates:

Factories for editors. createEditor() method can be reimplemented to return a specific widget that can edit a given cell in the model. It also has the gateway routines model->editor and editor->model for the data transit: setEditorData and setModelData. They are called at the beginning of the edit and at the end. It also provides the rendering logic for the element: method paint()

Supervising presenter

Supervising presenter is a simple partition of the Smart UI into logic and state classes. All state stays in the View, and all logic stays in the Supervising Presenter. The presenter observes the View for notification of GUI events, and updates the View through direct call. The View stays oblivious of the presenter, and acts therefore as a passive view. Testing is simplified because the View can be replaced with a mock.

The state remaining in the view is visual state. The supervising presenter extracts only visual logic.

FIXME integrate this pattern into the others, as it's clearly a duplication and can be rendered as part of Passive View or MVP.

FIXME: Difference with passive view is that in passive view the view is completely under control of the presenter, which drives all changes. In supervising presenter, the view can observe visual state provided by the presenter and react accordingly. It's a matter of degrees.

Presentation Model

A model, non visual representation of the view. It incorporates logic and state of the View. The View observes the Presentation Model and updates accordingly when events are reported. The model is unaware of the view.

Very similar in idea to Application Model. In practice, it is the same, just formalized.

FIXME: Integrate somewhere else?

Data binding

Motivation

We want to reduce boilerplate code to synchronize changes the bidirectional communication between the Model and the View.

The Model is a one-to-one representation of the View widgets. Changes in the content of the Model immediately propagates to the widget, and vice-versa, changes in the Widget are immediately propagated to the Model.

The code is declarative.

Validation is possible, but only for individual values. The Model can inform the View that the value is incorrect and raise an exception. This is interpreted by the View by coloring the cell red, for example.

synchronization can also be one-directional. Diretionality is important if we want to elimate coercion problems.

Data binding basically connects specific widgets with specific model attributes through a trivial, off the shelf controller that is invisible to us.

Possibility to say how to transform the data as they go back and forth. Backbone.js stickit onSet/onGet

The triad becomes a 1:1 connection between view and model, with the controller being invisible and generic.

Practical Examples

Cocoa Bindings, TraitsUI

Hierarchic MVC

Until now, we have seen MVC applied to a single triad. This works well for individual widgets and dialogs, but how do we apply and scale MVC to an application level? Communication between controllers

Application composed of tens, or hundreds of triads. Can we organize them somehow? Do we need to? Yes we do.

Controller hierarchy in Traditional MVC

In order to handle the user events, traditional MVC had to organize controllers in a hierarchy. When a View received an event, it was handed out to its controller . Events are delivered according to the actual cursor presence (which basically indicates which View is in Focus) until they rare given to a controller that is willing to handle the event.

only one controller active at a time.

Hierarchic Model View Controller (HMVC, Recursive MVC)

Hierarchic MVC is a strategy to apply MVC in large applications while keeping control of the granularity of data and communication. HMVC deploys a hierarchy of triads by connecting controllers. The triads work together by handling events they can handle, and forwarding them up in the hierarchy when they don't know how to handle

[IMAGE]

There are relevant differences when compared with traditional MVC: the view is responsible for handling user input events. These events are forwarded to the controller. The controller as usual performs modification on the model through direct method call on model objects To refresh the view state, the controller notifies the view that it needs refresh. The view then communicates directly with the model, pulling the data from it without involving the controller further. [4] Alternatively, the model notifies the view by providing its own state [5] The controller also acts as a hub in the controller hierarchy. If a controller receives an event from its associated view that cannot be handled, it is bubbled to the parent, which in turn can choose to handle it or delegate it further up or down in the tree. Any model at any level in the hierarchy can access data at any scope, and models can also talk to each other. Controllers at any level in the hierarchy can share these models.

controller has another controller as view controller handle multiple views or multiple sub controllers

View-model in a observer pattern

Presentation Abstraction Control (PAC)

PAC is an older scheme, very similar to HMVC. Similar to MVC, PAC defines a triad (Agent in PAC terminology) as follows: Presentation, responsible for handling all interaction with the user, both input (mouse events) and output (visualization) Abstraction, represents only data that are contextually meaningful within the triad. Control, connects Presentation and Abstraction, plus acts as a communication hub in a Control-connected hierarchy of Agents. Controls are responsible for forwarding the messages in transit in the hierarchy, eventually after transforming them. When an agent wants to send an event to another agent, it forwards it to its parent agent. The parent agent either handles the event or, if it does not know what to do with it, sends the event to one of its other children or to its parent,, and so on. At first glance, there's little or no difference between HMVC and PAC. If you think so, you are not the only one [4]. There are however certain important differences. First of all, HMVC is based on traditional MVC, meaning that there's tight coupling between the model and the view, with the view having to inquire the model. In PAC, this communication is fully mediated by the controller. With this strategy, PAC keeps MV loose coupling, while HMVC has MV tight coupling.
The second major difference is the scope of access of the triads. In HMVC, each triad is technically allowed to access all of the model. Not so in PAC. In PAC, it can only access data that is contextually meaningful to that triad. If it needs to access something that is not at its scope, it must forward an event to the controller, which will be routed to the proper context by the hierarchy.

PAC: groups of PAC entities, each one composed by a Presentation, Abstraction and Control object. organized in a network

- abstraction: represent the business functionality and state.

Control is a mediating controller. No interaction between P and A.

Advanced MVC

Until now, the main focus was on the basic patterns of MVC. In this chapter, we will observe the broader picture of a MVC application and how the patterns interact with the real world.

- Model persistence
- MVC Testing
- Notification Granularity
- Event Driven programming
- Thrashing prevention
- Validation
- Lapsed listener problem
- Publisher Subscriber
- Model Distribution
- Multithreading
- Notification looping prevention: Prevent recursive notification events to propagate.
- Delayed Model: Neutralizes fast notifications through a timeout.
- Throttling: Neutralizes fast notifications, but issue a change immediately.

Model persistence

Domain Model objects

In some cases, the model or part of the model must be made persistent (for example, to disk, or to a database) to be restored at a later stage. Persistency layer: sometimes it is considered part of the Model, but not necessarily

Which component should be responsible for the persistence? The most natural strategy is to let the model know how to store and retrieve itself from disk or database. This is a popular solution and goes by the name of "ActiveRecord". It is simple to use and understand, relatively flexible and intuitive, but it's not without limitation.

The first, and biggest limitation is that it favors strong coupling between the model and the IO strategy: abandoning the local disk storage in favor of a remote database will force us to reimplement the IO strategy of all the model objects;

A second problem is that Model objects lifetime is related to the storage backend. This makes testing the Model much harder, because the storage backend must be fully functional, or mocked;

The final problem of this approach is that, if the Model is fully in control of its persistence strategy, the client code cannot decide differently, for example, if it wants to store the model object somewhere else.

An alternative strategy is to delegate persistence to the Controller. The controller holds a reference to the model, and to a Storage subsystem. In response to proper trigger events, the controller can pick the relevant model objects and push them to the storage subsystem. This strategy has a few advantages: the model objects are lighter and know nothing of storage strategies, which can now be changed freely by using a different Storage service, potentially to a mock object during testing. The main disadvantage is that the additional flexibility requires more a complicated interaction. The storage can also be in charge of additional tasks, such as search and filtering of model objects, or creation (factory) of new objects, which the storage inserts into the database and hands out to the controller.

When it comes to data formats, there are many options, from the very simple CSV to the more complex like databases. A simple choice can be a nosql database, or a tinysql. Regardless of your choice, it's important you version your objects. The object is generally serialized in a stream of bytes and written to disk. Recovery implies de-serialization of the byte string, and reconstruction of the object last state.

ORM models

The model can be distributed over a network and accessed through proxy classes with none or minor changes to the remaining protagonists.

Persistency layer: aimed exclusively at services for persistence of models (so that they don't contain this logic themselves)

MVC Testing

Detail the challenges of asynchronous component in testing

A major problem resulting from a complex View is the difficulty of testing. Visual components tend to be more complex to test, because they require simulation of User interaction. Toolkits provide techniques to create events programmatically, but their handling depends on the event loop being executed. The blocking nature of its execution makes it impractical in a test setting.

Testing a GUI application is generally complex, due to the large number of interactions that the user may potentially choose to perform. Manual testing is tedious and error-prone, and simulating user events can open problems relative to their intrinsic asynchronous nature, and the visual nature of the result.

A sensible approach is to leverage the MVC structure to attack the testing problem in a programmatic, and possibly synchronous way.

In general, you should be able to test or even use the model independently of the controller and views. In fact, the model should be able to work without any controllers or views implemented at all. It's a completely separated layer with no dependencies toward GUI representation, widgets, or strategies to apply the changes. If this is not the case, then you have a code smell that needs refactoring. There are however exceptions to this rule [7]. For example, suppose your model is representing the state of a drawing program. The shapes that are inserted in the model are actual Shape objects that are graphical in nature, and it would make sense to assign to these objects the responsibility of drawing themselves on the view. This is practical, but it can backfire: it requires the objects to know about the specific view's details about how to draw itself onto it, meaning that a different view might not be compatible. Assigning representation responsibilities to model objects is a rare occurrence that can always be worked around, for example separating the mathematical description of the Shape (e.g. the corners of a rectangle) from the drawing logic (e.g. the actual graphic calls that draw the rectangle on the screen) and move this drawing logic in a Renderer class.

A view that acts on a widget knowing nothing about the model. View “adapter”

Mock view: it attaches to the model like a normal view, but has no GUI and can be probed. Fixture model: a model with well defined, well known data that can be attached to the view. The view should draw exactly what we know from the fixture model.

Microsoft Visual Testing Sending events with xtest, or with the widget level interface. Sporadics due to change in layout, running the screensaver.

asynchronous tests tend to be slow (you need to perform an action, then wait for the result that may come and sporadically broken, because of race conditions. Favor synchronous testing, restrict asynchronous.

Test of components that are hard to test should be minimal, and the behavior of these components should be minimal as well.

You can perform tests of graphical components by pushing events into the GUI toolkit event queue, but again, they tend to be brittle and asynchronous.

Watir, WatiN, Selenium

Interaction with the event loop

Until now, very limited mention was made about the event dispatch mechanism and the event loop: we remained oblivious of how UI events were delivered to the View/Controller. In the following pages we will examine how events are dispatched and how crucial is a proper coexistence between the dispatch mechanism and MVC.

In the most simple terms, an event-driven UI program is built on top of an *event loop*. This infinite loop generally performs these operations:

1. awaits for low level events, like a mouse click, a key press, or a request to show from the windowing system.
2. puts these events in a queue
3. fully consumes the queue and dispatches the events to the appropriate handler (e.g. a method on the widget currently in focus). In this phase the thread will traverse, among other things, your MVC.
4. Return to 1.

This loop is executed by the main thread of the program, which then
executes during a single iteration of the event loop can be extremely complex, and traverse the complex notification network of your MVC application. The event handler is, for all purposes, atomic. When it starts, it is the only part of your code that will be executed, and will run to completion with no chance for interruption. It is therefore critical, for the application to be responsive to subsequent events waiting in the next iteration of the loop, that an event is fully handled within 200 milliseconds.

The consequence of this requirement is that anything in your MVC code that blocks the executing thread will prevent the event loop to roll, and will slow down or freeze your application, offering a suboptimal user experience.

Examples of situations where the above may occur are the following:

- initiating a network connection (which may block until a timeout is reached)
- waiting for a state machine to switch state (e.g. become idle)
- a long running computation
- reading or writing a large file either from the disk or from a network connection
- running an external process that must be controlled, or whose stdout must be parsed.

You can see the event loop as a cooperative multitasking system. A cooperative multitasking system is a system allowing multiple tasks to run, provided that each task relinquish control when done, giving other tasks the chance to run.

To keep interface responsiveness, you must create a secondary thread of execution in your program, when the event handler expects to wait. The main thread spawns the secondary thread. The secondary thread executes the long running task, while the main thread can return to the event queue and keep processing events. This solution now faces the following difficulties:

- the secondary thread must notify its completion to the main thread, either successfully or with an error condition.
- the secondary thread must interact nicely with the main thread. Both threads could access the same shared state, for which synchronization is needed
- as a corollary of the above, the secondary thread should not call the event loop or any part of the code the main thread is fundamentally responsible for (such as UI handling) because it is probably not designed to be thread safe.
- the secondary thread could potentially trigger notifications through MVC (for example, by modifying a model state). The notification will propagate to View and Controller classes, which may involve UI, which again is not designed for being used by a secondary thread.

As you can see, handling multithreading in an event driven system is not trivial.

An alternative approach to what presented above is suspended execution. The concept requires the language to support suspending a routine execution and relinquish control back to the caller. When the suspended routine is reinstated, it will continue from where it left.

Suspended execution allows a handler to relinquish control back to the event loop, allow for other events to be processed, and restart where it was. This approach has the following drawbacks

- it requires language support (e.g. yield keyword in python)
- event handlers are no longer atomic. Processing other events might imply that the restarting handler is now handling a state that may no longer be consistent. A point of yield can be seen as a point where any code can be arbitrarily executed.
- Does not solve for blocking calls, but may allow for event processing if the suspendable task can be broken down into chunks, where one can periodically relinquish control (e.g. a loop).

This approach is equivalent to calling the processEvents manually at the point of yield.

For the blocking calls, the ideal solution would be to have a routine that acts as non-blocking and notifies back when completed. This is the case of a callback, a routine that gets called when the secondary thread has completed its task. The most common problem of this approach is that the callback will be executed in the secondary thread as well, so in general it cannot perform any action that may conflict with the main thread. The recommended

course of action is that the secondary thread, now running the callback, notifies the main thread through an event. Event queues implementations are generally aware of this need and are therefore thread safe.

MVC was considered as an independent design approach without much consideration of the event system. Strictly speaking, MVC does not require an event loop, but its usefulness would be severely limited.

Explain how MVC naturally ends up with an event driven model. Explain the complexity of debugging (e.g. backtraces all coming from the event loop)

Behavior is no longer characterized by code alone. Emergent behavior arises from the potentially asynchronous interaction among objects, communicating through events. The communication network being mutable. This results in extremely complex, hard to understand, hard to debug designs.

solve this? One solution is to spawn another thread, and let this other thread do the heavy, long running work, while the main thread goes back to the event loop and keeps processing events. But multithread programming is hard, and event driven multithread programming even more so. So ideally, you would prefer to have a single thread, but when it encounters something that is long running, you use a trick to suspend the execution point, and resume it later, when the long running task is completed, or maybe you want to do it in steps, each one short, but taken together they run for a long time. the yield keyword, and therefore generators, happen to provide this exact service: suspend something and resume it later from where you left it. When you reach the yield, the main thread can now go back to the event loop, and keep processing events. Somehow, where the execution was suspended to wait for a long running thing there will be some magic so that when the long running thing is done, the thread will know about it and go back to where it was suspended. This can be achieved either by a secondary thread, or by the main thread itself: when it runs out of events to process, it works on the long running task, maybe to be suspended again a little later. What you see is what is known as collaborative (or cooperative) multitasking. The event loop is basically a "kernel", and yield points are equivalent to "system calls" into the "kernel". At this yield points, the control is returned to the kernel, which is now free to run something else, interleaving all the handling and keeping the event processing alive, instead of being stuck at one particular handler. Note that this mechanism requires collaboration: the individual handlers must yield to inform the "kernel" "I'm not done here yet, but give someone else a chance to keep going". This compares with preemptive multitasking where the kernel is the one saying: "that's it, you had enough fun, let's someone else go now", which is what modern real kernels do. So asyncio is a form of event loop and collaborative

multitasking to allow event driven programming without either having an unresponsive application or having to deal with multiple threads, callbacks and all the horror that arises from it in an event driven environment.

To prevent trashing with many notifications, there are three strategies:

```
- disable notifications, to the operations, re-enable the notifications.
  this has the disadavantage that you might not know what notifications to
  send when they are re-enabled. One solution could be to spool them,
  and at re-enable, merge the duplicates and send out the minimum.
- have coarse grained operations, operating on large sets and sending out
  only one notification at the end.
- Have fine grained modification routines with an option notify that allows
  to decide when to send the notification and when not to.
- Have the model be a centralizer of the notification delivery, but have notifyObserve
r called
  externally.
- have a smart signal that can be put in a "trasaction on" mode, and accumulates the
  notifications, and then release the notification when a "commit" is issued
```

Validation

Who performs validation? View? Controller? Model? Consistency of the data inside the model? Depending on the application, the model can host invalid data (that is, invalid for the application)

Lapsed Listener Problem

A notification system introduces a potential for memory leaks known as "Lapsed listener problem". It occurs when a listener registers to a notifier, then goes out of scope without unsubscribing. The listener is never garbage collected due to the permanence of the notification connection. It is technically still receiving notifications, which may introduce additional problems if these notifications are expensive to honor. In languages without GC, if the listener is deleted, the notifier can now hold a reference to freed memory, potentially resulting in a crash.

There are various options to solve this problem. The first is to make sure the listener is correctly unregistered before going out of scope or released.

The second option is to have a notification system using weak references.

problem can exist also with callbacks that are closures, or when exceptions are stored.

Event bus / PubSub

Full decoupling between publishers and subscribers. Publishers don't know about subscribers, and vice-versa. Messages are sender to receiver. The receiver cannot "reply" to the sender.

Example, wxpython provides a nice example of pubsub model

```
import wx
from wx.lib.pubsub import Publisher

class Model:
    def __init__(self):
        self.myMoney = 0

    def addMoney(self, value):
        self.myMoney += value
        Publisher.sendMessage("MONEY CHANGED", self.myMoney)
```

Objects that are interested in the notification can now subscribe to the qualified notification as follows. The handler will receive a message, qualified with the appropriate information

```

class Controller:
    def __init__(self, app):
        # <...>
        pub.subscribe(self.moneyChangedHandler, "MONEY CHANGED")

    def moneyChangedHandler(self, message):
        """
        This method is the handler for "MONEY CHANGED" messages,
        which pubsub will call as messages are sent from the model.

        We already know the topic is "MONEY CHANGED", but if we
        didn't, message.topic would tell us.
        """
        self.view.setMoney(message.data)
```

NSNotificationCenter is a pubsub.

- decoupling makes compile time checks useless.
- delivery network can become complicated. Incorrect setup can lead to unintended listeners to receive messages they are not supposed to receive. the application flow is hard to understand and debug.

- from the code alone, it's hard to spot the dependency between a publisher and a subscriber, in particularly when the message is emitted and delivered.
- delivery can be synchronous or asynchronous, but even when synchronous, it's not possible to rely on delivery order.
- message source may not be available to the receiver.

use of topics to group message types.

Model Distribution

sharding?

Multithreading

Problem with multiple threads. Sending notifications that are delivered as the same thread. Describe how Qt manages to handle delivery through the event loop if two objects have different thread affinity.

Do not spawn threads. makes things harder to handle. use a thread pool.

Models should be synchronous, so you can decide which threading strategy to use.

Have futures.

Separate threads can act independently, produced by code running in the main (event loop) thread. The problem is that any change they can do can propagate through the network, and touch parts of the code that is currently handled by the main thread. As a result, the generally better way of handling this situation is that secondary threads communicate with the main thread in two ways:

- setting state (using locks for synchronization)
- posting events into the event loop, so that the main thread can handle them.

Once an event is triggered, the application has around 1/60th of a second to return control to the event loop, meaning that the object/notification network traversal must be over quickly. If any event triggers something that can potentially last for more than the mentioned amount of time, it must be executed in a separate thread, or the interface responsiveness will suffer. Having a separate thread carries its additional quirks: can't normally touch ui code, must be synchronized.

FIXME do comparison with asynchronous single threaded programming. Put a useful picture

Notification looping prevention

Notification messages from the Model can become problematic for a series of reason

- the Views get informed that changes occurred, but it's in a part of the data model that is not represented by a specific View. Views must go through a refresh cycle even if no data has changed for them
- A sequence of changes is performed on the Model, forcing a refresh of all the Views at each change, while a single refresh at the end of the sequence would suffice.
- The update-change cycle lead to an infinite loop

Consider the following case of a SpinBox containing the value 3, and the associated Model value currently set to 3 as well. When the user interacts with the SpinBox, clicking the up arrow, the following sequence of events occurs:

1. a valueChanged() signal is issued by the SpinBox with the new value, 4. We assume the SpinBox keeps showing the old value, as it represents the Model, which at the moment contains 3.
2. the Controller.setValue(4) method is called, which in turn calls Model.setValue(4).
3. the Model stores the new value 4, then issue a _notifyListeners to inform all the connected views, including the SpinBox.
4. the SpinBox receives the notify(), which now fetches the new value from the Model and sets the new value using QSpinBox.setValue(4)
5. the SpinBox is still containing the value 3. QSpinBox.setValue(4) triggers valueChanged() again.
6. Controller.setValue is called again, reproducing the situation at point 2.

With this scenario, the application is potentially entering a notification loop. A prevention strategy is to have the Model notify the listeners only if the new value differs from the currently stored one. This solution will terminate at point 3, technically performing useless Controller.setValue and Model.setValue calls. A tempting alternative solution is to have the SpinBox increment its visualized value independently from the Model, thus having the View autonomous in its visualized state. With this approach, after step 1 the SpinBox will show the number 4. The chain of events will unfold exactly in the same way until step 4. The SpinBox will now observe that the new value in the Model is the same as the one it is currently displaying, terminating the chain by not triggering a valueChanged(). Depending on the toolkit used, graphical Views may or may not behave as described, but the fundamental issue with this approach is that the View is assuming to know the next value, and setting it

accordingly, without involving any logic from the Controller or Model. The Model could, for example, consider the new value 4 to be invalid and set itself to the next valid one, for example 27. This will force the View to update its graphical representation again.

Another strategy is to prevent the View from updating itself twice within the same cycle of events. A possible implementation of this strategy is to hold a flag updating on the View. The flag is set to True at step 1. The chain of events develops in the same way until step 5, where the setValue operation will check for the flag. If true, it will only update the graphical aspect of the widget, and skip the triggering of the second valueChanged() signal. Another strategy is to have a View that does not triggers valueChanged under certain conditions.

Shut down the Model notification system? not a good idea. other parties will not receive events. Another alternative is to detach the View from the notification. It will not receive update notifications from the model, just set the value. It won't see changes in the model that originate from outside though.

To prevent notification trashing, one can rely on transaction, to turn off notifications on the model, perform a set of changes, then triggering the notification by closing the transaction. When multiple independent modifications must be performed on the model in sequence, it pays off to have a method to disable and enable notifications. Without this technique, every individual change would trigger an update and force a refresh of the connected views, potentially ruining performance and exposing the user to progressive changes through the interface as each change is applied. By disabling the notifications, performing the changes, and re-enabling the notifications, a single update will be triggered. model packing multiple changes to deliver a single refresh to the view controller disabling notifications of the model.

FIXME: Another example:

a view has two methods, one that stores stuff on the model from the widget content, and another one that takes data from the model and stores it in the widgets. If the widget.setValue(model.value) triggers a notification and a request for syncing (as normally happens when the user writes a value), we need to disable notification when the modelToWidget() method is called, otherwise it will trigger notifications into the model, potentially calling modelToWidget() again.

We normally skip this with a flag in the modelToWidget() method that prevents recursion by bailing out if set to true, and it's set to true immediately, or disabling notification for the widget to synchronize. Make a code example for this

Delayed Model

FIXME: Asynchronous. Move this one to advanced patterns.

Motivation

Every time a Model changes, the View must refresh against the new data. This step can be time consuming. If the Model is going through a lot of changes in a very short amount of time, shorter than the time needed to refresh the View, we might not want the View to follow through. This mechanism is known as "debouncing".

We can neutralize these fast changes in the Model either View-side or Model-side.

Design

The Model holds a timer. Every time a change is performed on the Model and the Timer is not running, the Timer is started. No notification is issued to the View until the Timer runs out.

Being the timer asynchronous, particular care must be taken to guarantee that the event is not delivered when the Model is undergoing another change.

In both cases, if the new change overlaps with the previous one, the old change can be discarded (as it will never get to appear on the view). Otherwise, the two changes can be combined if the notification is qualified. If it's not qualified, then when the View is finally notified at the end of the timer, it will get the current state.

Throttling

Motivation

Similar to debouncing, but the notification is issued immediately, and then not anymore until the timer expires. At the end of the timer, however, a check must be performed if the current value is different from the value issued at the first notification. If different, a new final notification must be issued, otherwise the View would sit desynchronized from the Model.

MVC Implementations

References [8]

iOS

In iOS and cocoa, the MVC is a Model View Adapter style. Coordinating controllers vs. mediating controllers.

delegates outlet data source notification and Key-Value Observing (KVO)

Java Swing

Model-Delegate Microsoft MFC

MVC On the web

Explain uniqueness of HTTP.

On the web, we have two entities interacting: the server and the client. The server is the only responsible to hold the model. The client runs a web browser that performs requests to the server. These requests are channeled properly, trigger changes in the model, and give origin to a response that the client renders to the user. Some interaction (e.g. scrolling a list of entries) may not involve the server at all.

On the web, the controller is responsible for handling user events, preparing the view, and pushing it to the renderer. a quite different pattern.

MVC can also be used on the web, and plenty of web development frameworks provide for free a well designed MVC architecture, where the programmer has just to fill the empty spaces and all the web heavy lifting is taken care of. As usual, the model objects represent the business domain of our application, with the task of persistence given to the model or to another layer: they either talk to a database directly (generally, but not always, an SQL one) or through an Object-Relational Mapper to convert the Object Oriented nature of the model classes into something a relational database can digest. Similarly, the Controller receives the HTTP request from the user, as dispatched by the Web framework. It is responsible for applying business logic and coordinating the other objects to display the final web page (or parts of it) to the user. In general, what the controller does at this point is to parse the request, selects the proper model objects to honor this request, selects a view appropriate for the request and let the view the task of rendering the final HTML for the browser's consumption. To do so, the view normally combines the controller-provided data with a template mechanism.

On the web, the separation between the View (the HTML and the browser as a renderer) and the Controller (the server side of the code) is strong and with a bottleneck in communication. Primary events are HTTP requests (GET/POST)

The controller can switch model or views as it sees fit, but in general it is initialized and deeply related to the view. How the controller get to know these different models or views can be done either through accessor methods (external code "pushes" the new model to the controller) or through a provider class (e.g. the controller knows where to get a model: the provider class hands it out when needed). knockout.js

Something about REST in web mvc

.. image:: web_mvc

View: renders the HTTP response from a template and the contents of the model. Controller: Handles the HTTP request as passed by the front controller, and selects the most appropriate models and View for the rendering.

Page Controller

To keep the view and the controller synchronized, there are two possible approaches: “push” strategy: data is pushed by the controller into the view. “pull” strategy: data is pulled by the view from the controller

For example, suppose the user adds a comment to a forum. Once he submits the request, its comment is now accepted by a controller, which will add it to the model. The view and the model are now desynchronized. The controller now can reply by pushing the new information to the view, so that the user-submitted comment can appear. Any other comment that was added to the model will also be pushed into the view, allowing the user to see its view change as comments are added.

In the pull model, the view is responsible for requesting and fetching data from the controller at the end of the submit request, and synchronize its content. Pull is also generally used to fetch any kind of data from the controller in response to a user request.

spring

ruby on rails

wxpy

On the web, the View is delivered to the client side for rendering in the browser, and the Model stays on the server side. When the User performs an action, the Controller will issue a change request to the Model, followed by a request to the View to refresh itself. The View will now issue a get request to the server to synchronize with the new Model contents.

User events as http requests, produced through either direct call or through XMLRPC calls

Page controller : handles requests for a specific web page

Front controller: Handles request for multiple pages.

The browser can be interpreted as a View: the page controller receives an http request and renders a html result, but it can also produce json, or xml, or any other format. this result is then sent to the browser for visual rendering. The controller selects the proper "view renderer" and may switch according to different constraints (e.g. having to present a page for mobile vs browser)

Backbone router

Original implementation of Smalltalk MVC: https://github.com/petermichaux/maria

Request model: The HTTP request coming in can also be seen as an object part of the model layer. Its change notifies the front controller, which acts on it

Two controllers in web mvc: Controller 1: mapping urls to a request handler, eventually using middleware to process the incoming request. Controller 2: the handlers.

The model never computes html

The view uses a template rendering. The controller combines state from the Model with the template engine to produce the resulting View representation.

Model: The server is responsible for handling potentially concurrent modification requests to the same model data coming from different clients.

Browser

A browser is fundamentally a MVC triple: it has a model (the HTML document) a View (the rendered content) and a controller (the part of the browser that interpret user events and eventually modifies the Model's content

Server side MVC

Most simple form of MVC. The client issues a request as a GET http request, eventually with POST data/cookies. The server handles the request and returns a full HTML page. Very coarse grained, very all-or-nothing interaction that forces the client to refresh the visual aspect often, instead of incrementally. The application is bandwidth hungry and sluggish.

Rich internet application

On the other end of the spectrum, we have RIA. RIA keep the model on the server, and move everything else to the client. The client model is synchronized with the server model. No rendering is performed by the server. it just returns data to the client.

sometimes referred , when in desktop applications, as "proxy delegate"

Other

Validation is normally performed twice: on the client to ensure the data is consistent and presented properly to the user. Once the user submits its data, though, validation on the server must also be performed. The request may be forced, and all the constraints we set from our View will be bypassed.

.. toctree:: :maxdepth: 1

Front controller

The front controller (sometimes called Application) is the first point of entry for the request. It handles the overall dispatch to the relevant controller, and deals with common needs such as authentication, session management, security, redirection. It contains logic that all requests must consider, reducing code duplication. It also adds data to the execution context.

Dispatch to the front controller is performed directly from the webserver.

Typically, the front controller can host a list of intercepting filters, classes aimed at performing specific operations on the incoming request.

A front controller can be a potential bottleneck. All requests pass through it.

FIXME: different interpretation of front controller. a single point of entry that replace the page controller design. The front controller creates commands (which replace page controllers) according to the submitted request, and these commands are executed by the front controller. Commands feed information to the views.

FIXME: django approach is kind of mixed. There's a front controller that dispatches to methods that in practice become page controllers. Before the dispatch happens, the request is passed through filters (middleware)

Page controller

A page controller handles a specific request on the server, by combining model and view and send back to the client the rendered view. Normally, it is bound to a specific URL.

It normally handles one logical page, or one specific action. This makes it extremely simple.

Once it receives the request, it extracts data from this request (e.g. HTTP headers, query parameters, cookies, form contents), performs operations on the model, then select the proper view to perform rendering. A template mechanism is generally used to render model data while separating rendering/presentation logic. It can render a whole page, a subset of it, or data that are parsed by the client-side javascript.

Testing the page controller can be complex due to the need to simulate an HTTP request. Ideally, one would partition non-HTTP related functionality and make them testable without an HTTP context, otherwise the web framework environment must be present and set up.

The page controller often is implemented as a reimplementation of a base class, to provide reusable functionality that is common to all page controllers.

In general, functionalities like session management, authentication, and similar low-level operations are not handled at the page controller level. A different technique is used: Filters (aka Middleware)

FIXME: common functionality between different page controllers may require inheritance. This can force hierarchies that not necessarily are appropriate, and force refactorings as new pages are added.

Templated View

To render the contents of the model, normally but not exclusively in HTML, a template View is used. The template is a blueprint document, containing HTML mixed with placeholder tags. These tags are replaced when the controller applies model's data to the template, producing the final HTML content.

This mechanism isolates the data from the production of HTML. It also allows to select a different template if the same data must be delivered in a different form (e.g. JSON, or XML) or for a different device (HTML for a computer vs. for a cellphone). Template authors don't need to know the framework, but they can focus on the visual result, while controller authors don't need to worry about visual issues, and focus instead on passing the needed data to the template.

Middleware filters

There are operations that are common and require a lot of pre or post processing in web handling. Things like authentication, session management, compression, cookie extraction, logging, and so on. It's common to provide the code dealing with these issues as pluggable components called Intercepting Filters or Middleware. These operations are common to all pages and it's convenient to centralize them. They also tend to be agnostic of the actual application, therefore offering a huge reuse potential.

Filters are generally not dependent on each other. When they are, it's important that they are traversed in the proper order.

They act as pre-processing filters of the request and post-processing filters of the response.

Considering that all requests traverse these filters, their performance is of the highest importance.

[FIGURE: put a layered request -> f1 f2 f3 controller -> response -> f3 f2 f1]

Javascript MVC

The client-server nature of web programming gives raise to the need for MVC on both sides. While we examined MVC on the server side until now, we now examine client side. This is normally done through javascript.

Views

JavaScript views are normally concerned with manipulating

- the DOM, or
- a canvas object

FIXME Maria implements true MVC. All other frameworks implement a derivation of MV*, where in particular the controller takes a different approach. Backbone uses Routers. Study javascript mvc in more details.

backbone synchronization with CRUD operations against a JSON/REST model.

Questions and Answers, Tips and Tricks

Should I use a simple container approach (e.g. key/value dictionary with notifications) as a model, instead of a full-fledged object?

A simple container like a key/value dictionary can technically be used as a model, often enriched with notification features. Typically, these Models have an agreed convention on the key, normally a string. This solution is fine for trivial Models, but as the amount of stored data grows, its unstructured nature and mode of access will lead to entangled, inconsistent and undocumented storage. The model will be a “big bag of data” with very little clarity or enforced consistency.

```
Enforcing access through well defined object relations and interfaces is
recommended for models beyond the most trivial cases.
```

How to report errors in the View?

```
This is more of a Human Interface design question, but the choice can influence the
design choices at the level of Model and View. It also depends on the data. Individual
values that are incorrect can be marked in red. Typically, the user would input some d
ata.
the value would be checked for validity, and if found invalid, the View would be chang
ed
to express this information. This case probably enjoys using a Local Model, so that ch
anges
can be discarded and the original Model is left untouched. It also means that the Mode
l
must be able to accept and preserve invalid data, because this invalid data
may be a step stone to reach a correct state after additional modifications.
```

Notes

the notification mechanism basically allow to inject code in setters.

Local models, or one global model?

the view should be able to query or inform the controller for action (will, should, did): delegate

Compare OSX bindings with MVVM ? Same stuff? FIXME: relationship with apple, where the view is replaced with the child controller ?

All intralayer communication can only be routed through controller-controller connection. This connection is bidirectional.

following the hierarchic composition of the GUI nesting. The model can be the same. In pratice, the scheme given above can be simplified by assuming a given hierarchy talks to the same model In J2EE, this approach is also known as Composite View.[11] [PIC of an example of a hierarchy with real widgets]

Keep networks simple and close.

Order of notification can have unexpected consequences on the state of your program.

keep your views able to deal with missing model or invalid data.

-

FIXME: pluggable view is overloaded as a term. Detail.

Naturally evolving toward a more declarative style

define your models as having atomic operations, be careful of uncontrolled notifications.

Problem with double notification if one notification is a subset of another. e.g. contentChanged/lineMetaChanged and contentChanged/lineAdded. How to handle the double notification? Pass an "event id" in the signal so that the client realizes that it's the same change that delivers two messages?

FIXME: MVC Notifications are always considered to be synchronous. They invoke methods directly and wait for them to return.

Typical situation atapplication startup, a communication network is created. the event loop takes control, in the main thread. Every time there's an event, the event loop dispatches that event. This in turn triggers changes that obey the static network and the dynamic network

through either direct calls or indirect (notification) calls. During the whole cascade, the application is frozen until control is returned to the event loop.

Distinguish between Notifications (that are synchronous calls using the observer pattern) vs events (that instead are dispatched through the event loop)

With signals, you might have to adapt the signals that your model emits to the specific needs of your views. A coarse grained signal that forces a heavy refresh on the view may be better split into a separate signal specific to the area of the model that actually affects the view. In practice, the model communication pattern may have to adapt to the View's implementation details to guarantee responsiveness.

For example, if you have a view displaying the number of lines in a document, subscribing to a contentChanged signal may require a recalculation of the number of lines at every character inserted. It may make sense to provide a lineNumberChanged signal, so that line number display is updated only when the model actually performs a change in the total number of lines.

Some toolkit and libraries allow you to define the network dynamically. Other allow you to define the network statically. The network cannot be changed.

How to handle null values.

Practical Example: Monitoring file contents

A Model that represents the contents of a file is unable to trigger any notification if changes from an external source are performed in the file. [1] the Model can be implemented so that the Controller occasionally requests (e.g. with a timer) to verify if changed occurred. When such changes are found, the Controller can act to update the View. Alternatively, the Model can perform notification to the listening Views as in an Active Model

- or by having an Active Model that uses a secondary thread to monitor for changes in the file, and issues notification when such changes are found

unless an appropriate implementation handles

[1] In this discussion, we ignore the problem to obtain a consistent state while reading a potentially incomplete file.

References

Pluggable View

- The Smalltalk MVC paradigm with pluggable views - Tong Sin Yin, Chow Pui Yee

PubSub

- Why NSNotificationCenter is bad

General

- Model-view-controller architectural pattern and its evolution in graphical user interface frameworks - Matti Bragge